# AN UPDATE GATHERING ARCHITECTURE FOR DATA WAREHOUSES AND EXTERNAL TRIGGER PROCESSORS

By

TONY CHRISTOPHER CARNES

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1999

Copyright 1999

by

Tony C. Carnes

Dedicated to my wife, Betsy, and my children, Joshua, KC, and Maggie.

ACKNOWLEDGMENTS

Eric Hanson, Associate Professor at the University of Florida, deserves my utmost gratitude for his assistance in suggesting content as well as editorial input. Without his help, this paper would not have been possible.

John Morgan and Glenn Travis, two of my former colleagues, have been quite instrumental in helping solidify the background knowledge necessary to formulate some of the algorithms developed here.

Finally, I thank my wife, Betsy, and our children, Joshua, Kenny, and Maggie, for their patience as this process has kept we away from them far to often.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS ..... iv

LIST OF TABLES ..... vii

LIST OF FIGURES ..... viii

ABSTRACT ..... ix

CHAPTERS

1 INTRODUCTION ..... 1

2 RELATED WORK ..... 4
Active Databases ..... 4
Gateways/ODBC/OLE DB ..... 8
Replication ..... 10
Differencing ..... 13
Wrappers ..... 14
TriggerMan Project ..... 15

3 TRIGGERMAN'S DATA SOURCE ARCHITECTURE ..... 19
Data Source Definition Language (DDL) ..... 19
Basic data source architecture ..... 22
Data Stream Data Sources ..... 30
Replication Based Data Sources ..... 32
Trigger Based Data Sources ..... 33
Differencing Based Data Sources ..... 35
Local Data Sources ..... 36
Application Based Data Sources ..... 36
Application Modification ..... 37
Middleware Modification ..... 37
Queriable vs. Non-queriable Sources ..... 38
Transactionally-Consistent Update Propagation ..... 39

4 DIFFERENCING-BASED CHANGE DETECTION ..... 44
Differencing Issues in OLTP Systems ..... 49
Query-Based Differencing ..... 51
Modified Database-Schema Approach (MDS) ..... 54

Modified Table-schema Approach ..... 59
Simple Time-Stamps (TS) ..... 60
Time-Stamps and Delete-Flags (TDI) ..... 61
Tuple Level Checksums (TLCS) ..... 66
Incremental Differencing ..... 68
Local Site Incremental Differencing (ID) ..... 70
Remote-Site Incremental Differencing with Checksums (ID/RCS) ..... 72
View Based Differencing (VBD) ..... 74

5 PERFORMANCE EVALUATION ..... 78
Varying Size of Replica ..... 79
Varying the Number of Changes Occurring Between Iterations ..... 83
ATDI Varying Change % and Distribution of Modifications ..... 84
MDS Varying Change % and Size of Duplicate Relation ..... 90
Performance Summary ..... 94

6 CONCLUSION ..... 97

LIST OF REFERENCES ..... 100

BIOGRAPHICAL SKETCH ..... 106

LIST OF TABLES

Table page

1. Augmented Schema Definition ..... 28
2. Update Descriptor Definition ..... 28
3. Definition of Terms ..... 53
4. Definition of Variables with Default Values ..... 54
5. Qualitative Comparison of Update Gathering Techniques with Respect to Selected Relevant Issues ..... 95

LIST OF FIGURES

Figure page


1. Rule Syntax for TriggerMan and SQL3 ..... 7
2. Sample TriggerMan Multi-destination Update Trigger ..... 8
3. Two-tier ODBC architecture with a middleware vendor ..... 9
4. Typical Replication Architecture ..... 11
5. TriggerMan's software architecture ..... 16
6. TriggerMan's architectural platform ..... 18
7. TriggerMan's data source architecture ..... 22
8. Data Queuing within TriggerMan ..... 25
9. Basic Process Layout for GDS ..... 31
10. Application Modification Using Data Source API ..... 37
11. Typical Distributed Transaction Communication Architecture ..... 43
12. Simple Query-Based Differencing Algorithm (QDA) ..... 55
13. Time-stamp with Delete-Flag Algorithm ..... 62
14. Performance Analysis of TS&DF Algorithm ..... 63
15. ATDI Algorithm Costs using a secondary index ..... 66
16. ATDI Algorithm Costs using an augmented secondary index ..... 66
17. ATDI Algorithm costs in conjunction with a Projected Copy ..... 66
18. Basic Incremental Differencing Algorithm ..... 69
19. Illustration of Remote-Site Incremental Differencing (RID) ..... 72
20. Varying % of Base Relation Tuples found in Replica ..... 80
21. Varying % of Base Relation Attributes found in Replica ..... 80
22. Varying Replica Size as a % of Base Relation ..... 81
23. Algorithm Comparison Using Full Duplicate varying the number of changes issued between iterations ..... 83
24. Comparison of ATDI1 and ATDI2 Algorithms ..... 86
25. ATDI Version Comparison Varying Type and Distribution of Updates ..... 89
26. Performance of MDS Algorithm Restricting Replica Tuples and the Number of Changes Applied between Iterations ..... 91
27. Performance of MDS Algorithm Restricting Attributes of the Replica and the Number of Changes Applied between Iterations ..... 92

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

AN UPDATE GATHERING ARCHITECTURE FOR DATA WAREHOUSES
AND EXTERNAL TRIGGER PROCESSORS

By

Tony C. Carnes

May 1999

Chairman: Dr. Eric Hanson
Major Department: Computer and Information Science and Engineering

The introduction of external trigger systems and the popularization of data warehouses have served as a catalyst for developing efficient change detection techniques. The update-gathering architecture presented here utilizes several new algorithms and a robust communication facility to efficiently isolate changes in queriable data sources and deliver the update descriptors to appropriate destinations. Much of this work was developed for use with the TriggerMan asynchronous trigger processor project, but it has widespread applicability to other areas. Efficient isolation and delivery of update descriptors can also be used to reduce the re-load time of data warehouses, to enhance data security, and to facilitate data replication.

In this work, we propose multiple change detection algorithms as the optimal approach is determined by the capabilities and change patterns of the underlying data source. We focus on simple and legacy data sources that support querying but do not

necessarily support other methods that allow for more efficient change detection, such as simple triggers, application modification, log readers, or asynchronous replication. We argue that most traditional sources of data will have some structure and are generally queriable through some form of SQL-enabled gateway. In addition, we propose that some information is known about the content of the data as a replicate copy is created before comparisons begin. Utilizing these facts, our work improves upon previously developed differencing algorithms by "pushing down" restrictions (reducing I/O), by comparing only interesting attributes (reducing CPU), by incrementally comparing subsets where necessary (reducing locking), and by eliminating a copy step from traditional differencing algorithms. Combining these techniques enables frequent differencing on data sets of all sizes. The isolated sets of change descriptors can then be used to update snapshots in a warehouse or they can be used to determine which ECA rules should be activated.

# CHAPTER 1
# INTRODUCTION

Efficient methods for isolating update descriptors in heterogeneous systems have been desired for many years. Detecting changes within a data source can be used to enhance data security, to facilitate active capabilities, and to efficiently propagate data into a data warehouse. The TriggerMan [Hans97] external trigger processor project has served as a catalyst for designing highly efficient, readily implementable change detection algorithms. The algorithms proposed in this dissertation are designed for use with standard data sources that are compliant with Open Database Connectivity, ODBC [Penn97]. There are over 130 different data sources that are currently accessible through an ODBC gateway. In addition to the algorithms, the TriggerMan project has led to the creation of an update delivery architecture that can efficiently guarantee update descriptor delivery and provide for transactionally consistent update processing in a heterogeneous environment. While the data source architecture is specifically designed for use within TriggerMan, the algorithms can be readily adopted for use in many other contexts.

TriggerMan's extensible architecture provides for asynchronous trigger processing and data sharing without the limitations inherent in previous technology. As such, TriggerMan provides for data sharing, complex rule condition testing, and inclusion of non-DBMS sources of data as input to the rule condition tester. Due to the vast range of capabilities in different data sources, TriggerMan has been designed with an extensible architecture that allows utilization of many different methods for data gathering. Often,

existing applications can remain unchanged and yet still provide data to TriggerMan and thus indirectly to other sources of data through the utilization of these algorithms. The ability to gather data without modifying the application interface is especially critical when data is polled from legacy systems. The asynchronous, distributed nature of the architecture provides excellent performance characteristics when compared to transaction based rule condition testing and currently available push technology.

The TriggerMan data source architecture addresses numerous key issues. Efficient gathering of updates from both queriable and non-queriable sources requires the utilization of algorithms that are tailored to the capabilities of the underlying data source. Communication to heterogeneous sources demands the development of a robust interface. Propagating updates in a transactionally consistent manner requires implementation of an efficient architecture and commit protocol to provide transaction atomicity in a distributed, heterogeneous environment. Ensuring the persistence of update descriptors being passed to TriggerMan, and ensuring the propagation of update requests from TriggerMan requires implementation of persistent queues. If any issue is ignored, the entire architecture can fail.

The core of the work presented in this dissertation has been developed as a sub-component of the TriggerMan [Hans97] asynchronous trigger process project. Some aspects of the TriggerMan project have been included in this paper to facilitate understanding of the requirements for the data gathering mechanism. In addition to the TriggerMan project, a brief introduction to other areas related to data capture and propagation is provided including active databases, gateways, replication, wrappers, and

differencing. The limitations of each of these techniques, as it relates to update descriptor gathering, are also discussed.

The TriggerMan asynchronous trigger processor, as well as any data warehouse, requires the ability to receive data from many different sources. When a source of data is identified, the method used to isolate and propagate update descriptors is also selected. We explore the performance characteristics of nine different methods that have restricted use but that provide good data gathering performance in the appropriate contexts.

To ensure that changes can be detected in a more general context, we explore the use of differencing [Chaw96][Chaw97a][Chaw97b][Labi96][Lind86]. Several new differencing-based algorithms are outlined in the context of ODBC compliant data sources. The performance characteristics of each algorithm are explored and the scenarios in which each algorithm is most appropriately utilized are outlined. A performance comparison to previous techniques is provided to demonstrate the vast improvement in I/O costs that can be realized using the new algorithms.

The work outlined in this paper can be used to forward update descriptors to TriggerMan as well as to redesign the way data is applied to data warehouses. With the introduction of these techniques, the performance problems associated with locking, I/O, and CPU utilization have been minimized. The algorithms and techniques developed here ensure the delivery of update descriptors in a timely fashion without a significant reduction in concurrency for the data source. Appropriate use of these techniques will effectively eliminate the need for an eight hour reload of the data warehouse each night and they will enable the creation of an external trigger processor by identifying and forwarding update descriptors in a timely manner.

# CHAPTER 2
# RELATED WORK

There have been several attempts put forth to enhance the sharing of data between homogeneous and heterogeneous data sources. These past approaches include, but are not limited to active databases, ODBC/OLEDB, gateways, replication, differencing, and wrappers. While several solutions hold promise, a hamartia in each concept limits its usefulness for out purposes. However, some aspect of each these technologies contributes to the design of the data gathering mechanisms in TriggerMan and thus the basic premise behind each is outlined here. A separate section is provided for TriggerMan, as it's data gathering mechanism makes use of augmented versions of many of these techniques.

## Active Databases

Active database systems have recently come into vogue due to their potential for decreasing application complexity and to their promise of improving database application response times [Jaeg95][Hans94][Hans95][McCa89][Wido96a][Wido96b]. Used effectively, active databases with stored procedures can reduce network traffic in distributed systems by 50% or more by minimizing the need to communicate all requests to the server and by lessening the number of acknowledgements sent to the client. Procedures running within the server greatly abate the copying of data to client address spaces as well. A rule defined within a single DBMS can be used to determine if the data

should be shared, if additional actions must take place within the database, or if notifications should be sent to subscribers. Unfortunately, active capabilities are generally limited to commercially available relational DBMSs.

The premise of active databases is fairly basic. Predefined rules stored within the database are checked when the state of the data is modified to determine if the changes affect a specified data set. If a rule condition is matched, a trigger or procedure can be used to perform additional work as well as to initiate statement rollbacks if the state transition is found to be invalid. Rules, triggers, and internal procedures are typically stored in system catalogs within the database management system. Externally stored procedures can also be made available to the storage facility through the use of remote procedure calls (RPCs).

Many relational database management systems have incorporated some active capabilities into their products. INGRES, Sybase, Informix, and most recently Oracle have adopted the same basic architecture for enhancing their products with active capabilities [Brow97]. Research projects that have forwarded improvements in this area include Ariel [Hans92a][Hans96a][Hans96b], HiPAC [Chak89][Chak91], Postgres [Ston87][Ston90], Starburst[Wid91][Wid92], Datex[Bran93], Paradiser[Dewa92], PARDES[Etzi93], and Reach [Kud93]. Other projects, like Sentinel [Liao97], have begun adding active capabilities to object-oriented database systems (i.e. ObjectStore [Lamb91][Oren92]). The ACT-NET consortium has published a document outlining the expected functionality of a fully active system [ACT96].

One of the strengths of active databases is their ability to incorporate the trigger actions within the triggering transaction. However, the *necessity* of using coupled

[Chak89] activation is a hindrance for some types of transactions, especially transactions spanning distributed heterogeneous systems. The binding of the procedure action to the transaction can lengthen the duration of the transaction and render poor application response times if the active capabilities are not used judiciously. Seperating, or uncoupling the trigger action from the initiating transaction can be beneficial in many scenarios including the generation of email and the sending of updates to other systems. Indirect implementation of uncoupling is accomplished by signaling an externally written application to perform the action. The signaled application passes updates to other data sources or simply notifies appropriate users of the detected condition. Uncoupling of the procedure invocation from the transaction has been discussed theoretically, but has not yet been directly implemented commercially.

There exist several drawbacks to the external application approach, however. Separate applications must be written to achieve asynchronous running of procedures. In addition, all rules defined against any tables in the transaction must be examined *as part of the transaction*. Finally, the ability to simultaneously connect to several heterogeneous data sources is limited to only a few commercial products. If an application is notified of an event, the application may first need to check several other sources to determine if all criteria are met before invoking the procedure. If the contributing sources are heterogeneous, the writing of the application to poll and deliver updates to these sources will be quite complex.

TriggerMan solves many of these problems by allowing for the creation of robust triggers in an expressive, intuitive language that can request updates to multiple sources given a single matching criteria. As an example, figure 1 outlines the same trigger using

the rule syntax of TriggerMan and the rule syntax of the proposed SQL3 standard. As can be readily seen, the TriggerMan syntax provides a much more intuitive feel as well as a more expressive interface.

| TriggerMan Rule Syntax | SQL3 Rule Syntax |
|---|---|
| Create trigger <triggerName><br>[in <setName>]<br>from <dataSourceList><br>[on <eventSpecification>]<br>[start time <timePoint>]<br>[end time <timePoint>]<br>[calendar <calendarName>]<br>[when <condition>]<br>[group by <groupList>]<br>[having <havingCondition>]<br>do (<statement>,begin <statement><br>{;<statement.} end) | Create trigger <triggerName><br>(No Cascade Before, After)<br>(Insert, Delete,<br>Update <colname>[,<colname>])<br>[Referencing (Old [as] correl_name,<br>New [as] correl_name, Old_table [as]<br>Id, New_table [as] Id)]<br>(For each row, for each statement)<br>mode db2sql<br>[when <search condition>]<br>(<sqlStatement>,<br>Begin Atomic <sqlStatement><br>{,<sqlStatement>]} End) |

Figure 1: Rule Syntax for TriggerMan and SQL3

The rule syntax for SQL3 [Date93] is much more complex, requires the writing of procedural SQL when multiple updates are required, and provides no significant functionality that is not readily available in the TriggerMan's rule specification syntax. In fact, the SQL3 syntax does not allow for temporal [Sist95] or join triggers to be created, nor does it allow for updates to heterogeneous data sources as the binding of the action is with the rule firing. As an example, given two sources of data, acct and personnel, one an Excel spreadsheet and one an Informix database table, we want to update the accounting spreadsheet whenever the personnel department hires a new person. In addition, we want to keep a readily available total of the number of employees per department in the dept_total table. After defining the data sources, the trigger specification within

TriggerMan is found in figure 2. The location of the data sources are resolved by the meta-data stored within the TriggerMan catalogs.

```
Create trigger newhire
From personnel
On insert personnel
Do begin execSQL "insert into acct values (new.id, new.name, new.salary)";
        execSQL "update dept_total set count = count + 1 where deptId=new.deptId;
end
```

**Figure 2: Sample TriggerMan Multi-destination Update Trigger**

With the SQL3 syntax, this type of trigger is impossible. One aspect of the SQL3 syntax that is not included in the TriggerMan syntax is the ability to define whether the rule should fire before or after execution, even though it is still part of the same transaction. As TriggerMan is designed to run asynchronously, it is assumed that the action that triggered the rule condition check has already committed. The only area in which this is not desirable is if the rules are designed to ensure referential or domain integrity. In such a case, TriggerMan can undo the transaction or notify the appropriate user(s) that the issued update has left the data in a non-desirable state.

## Gateways/ODBC/OLE DB

In the late 1980s, many database vendors proposed gateways as a way to create a single front-end that could communicate with backends/servers from competitors. Gateways enabled smaller Database Management Systems to compete by lessening the need to select one database vendor after the merging of units that previously used different storage products. As client-server computing became more widely accepted, another attempt at creating universally usable front-ends was introduced by Open

Database Connectivity (ODBC). Realizing the necessity of connectivity, many database vendors developed and published a database server application program interface (API) that could be used by non-vendor-specific programs to communicate directly with the server. ODBC makes use of these APIs so that connection to different vendors' databases can be achieved by simply substituting a different intermediate layer ODBC (see Figure 3). The ODBC driver expects a specific syntax to be passed to its own API, performs any needed translation of the queries, and forwards the request to the database specific API. Results are returned and translated in the reverse direction before being passed to the user interface. Although some performance problems exist with ODBC drivers, they enable a company to switch storage vendors without requiring front-end rewrites. While reducing the cost of application maintenance is important, ODBC does not directly facilitate the sharing of data between heterogeneous systems.

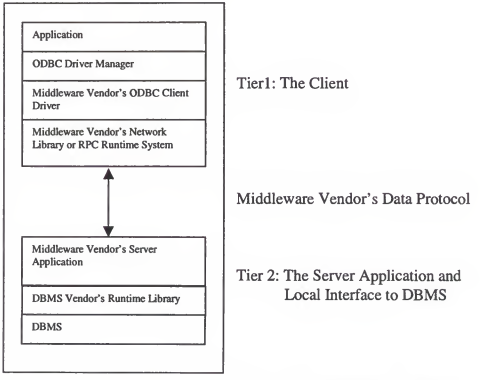


**Figure 3: Two-tier ODBC architecture with a middleware vendor**

Another significant drawback of ODBC is the necessity of writing front-ends in a language specifically designed for this layering approach. In addition, the sharing of data among multiple sources is limited to the application interface that uses ODBC for multi-source connections (available in version 3.0 ODBC) [Penn97]. With 80 to 90 percent of currently usable code written in a 3GL or a vendor-specific 4GL, other approaches enabling the sharing of data between heterogeneous sources are needed as requiring the rewriting of so many interfaces to enable their use with ODBC is not reasonable.

Some recent updates in the ODBC arena have been to include statement verification within the ODBC driver instead of forcing the database to perform all validation [Usof96]. This technology can lessen the load on the server and make better use of client/server architectures. The underlying difficulty of extending this capability beyond validation is the necessity of determining update commital before forwarding the information to a requesting source. These augmented middle-ware drivers provide some promise for efficiently detecting updates to non-relational systems.

## Replication

Another approach database vendors have provided to enhance sharing between like-kind data sources is replication. Replication is the process in which modifications to one database are copied to another. This can be performed either synchronously or asynchronously, but most application developers choose asynchronous replication. With full replication, the sites involved may be exact copies of one another when the systems are quiet. With partial replication, data can be horizontally and

vertically partitioned so that only subsets of data are copied to select sites. Each database receiving replicated data may also contain non-replicated data generated from applications acting on that site.

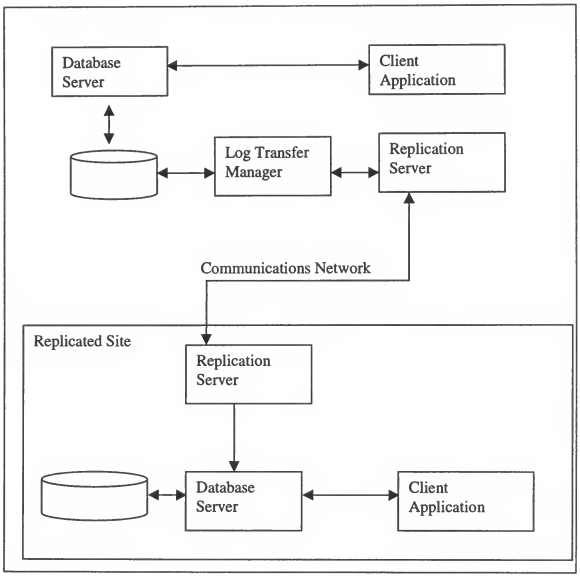


**Figure 4: Typical Replication Architecture**

In distributed environments, data can be asynchronously replicated to enable local site querying as well as to enable separation of OLTP systems from report-oriented ones. If data is to be replicated into a heterogeneous source, a gateway or a separate application must be used to forward changes that have been identified by the replicating database.

Figure 3 demonstrates the typical replication architecture for products such as Sybase, Ingres, and Oracle [Mois94][Comp94][Orac94].

Existing distributed transaction mechanisms and active capabilities within the DBMS provided early attempts at replication. The active-based approach led to poor performance but enabled synchronous replication. Vendors have recently modified their approaches and are accomplishing replication by using proprietary software to scan the log file for interesting changes. Scanning the log file is much more efficient as it does not lengthen the issuing transaction and thus places little overhead on the generating application. Although replication is useful, few non-DBMS applications support it. Lack of global availability seriously limits many of the scenarios in which it could be used effectively.

A serious pitfall of pure replication lies in the issue of multiple masters. Replicated data need only flow in one direction when changes are disallowed at all but one site. Few problems are introduced even when updates are allowed on disjoint data sets at multiple sites. Complexity is added when multiple sources modify the same data set and then attempt to replicate those changes to each other. As replication occurs asynchronously to prevent the need to perform two-phase commit, it is possible for differing sites to update the same tuple set before updates from the other site are received. In such situations a conflict resolution protocol must be provided by the vendor/vendors involved, or the users must write their own mechanism to resolve conflicts.

Basic replication also cannot be used directly in scenarios needing conditional replication. Although horizontally and vertically partitioned replication is now available in several products [Info97a][Ingr91][Micr96[Orac94][Mois94][Syba96], none directly

provide for replication based upon a condition in another database. An example is a central site that needs to be made aware of changes at distributed sites that have disjoint data sets. There is no known product that directly enables replication from site A to site B only if some criteria at site B holds true. In a manufacturing environment where products are manufactured at several sites, it is often beneficial to transfer materials from one site to another. If one site has received more orders for products than it can fill, a trigger can be defined to automatically issue a transfer request from site A to site B when additional product is available at site A.

Pure replication also requires a dedicated connection between relating sites. For distributed sites, there is often no need to keep an expensive dedicated line active from the central site to all participants if the update rate is low. A snapshot refresh approach [Lind86] can be more cost effective if remote-site changes are easy to identify. If the data set size is high, the cost of a pure dump and reload on the weekend may be excessively I/O intensive. Simple triggers can be defined in each remote site, if triggering is available, to send changes to a local application that could then forward changes to the central site, but this would entail writing an application for each such occurrence.

## Differencing

Differencing is the process of using an available "difference" function to compare two objects. The difference function can range from any operating system difference function, like the Unix and VMS "diff" command, to using SQL to compare fields using a "where" clause. In all of these cases, the data source provides some type of interface that can be used to glean information about specific tuples or attributes. Not all

sources of data provide such an interface, however. An example of one such data source is a manufacturing environment in which sample data is polled and passed directly to TriggerMan - a type of insert-only interface. Other data sources that need special handling are those that are not supported by ODBC or any other interface. If the data source does not allow direct querying for security purposes or any other reason, and if it provides no mechanism of directly discerning changes, one of several differencing techniques must be used to infer the desired information. TriggerMan's architecture is designed to drastically improve the performance of data change inference in non-queriable systems by enabling the storage of objects within TriggerMan. Storing a data state within TriggerMan eliminates the need to store and query the old state from each site. To further improve performance, the difference function selected is based upon the data structure and the capabilities of the underlying data source. A modification of this approach for data sources that provide a queriable interface is discussed at length in chapter 4.

Wrappers

Wrappers, which are not yet commercially available, are a technology designed to increase the ability of a user to define an access strategy to a data source. For traditional sources of data, an application programmer may select a gateway or ODBC driver that is specifically designed to allow access to a desired data source through the use of some query language. Gateways are designed to accept a vendor specific SQL dialect, translate it into a dialect or language understood by the data source, accept results from the data source, reformat the data, and then return the results in a format understood by the querying application. ODBC performs similar functions, but it is designed to accept a

specific ODBC language format instead of accepting any SQL style dialect from the program.

Wrappers are designed to enable a programmer to specify the interface to a new data source type by simply defining a few methods. Wrapper developers believe that an O-O approach is better than the ODBC one because no SQL-like language can fully specify how to gather data from all types of data sources. In addition, a multi-tiered design is implemented in some wrapper projects such as Garlic [Roth97] to enable join access to heterogeneous systems.

With Garlic, a user specifies a wrapper that is activated on the same machine as the data source, not unlike TriggerMan's Helper process. The Garlic server accepts requests, parses and optimizes the request to determine sources of data, forwards sub-requests to all pertinent wrappers, gathers information from those sources, then forwards the compiled results back to the requestor. This style is exactly equivalent to INGRES Star technology with Gateways [INGR91] or System-R Star technology [Lind86], except that it can be used for non-relational data with a less structured format.. The benefit of wrappers lies in the ability to quickly define interfaces to new data source types through the use of provided methods. The downfalls of this approach are the requirement of writing a new wrapper, which is essentially writing a new application program, for each new source of data as well the inherent performance problems of the architecture.

## TriggerMan Project

The perceived need of an external trigger processing system was the driving force behind the development of the TriggerMan project. With most non-DBMS sources of data having no triggering capabilities and with corporations having heterogeneous

systems that need to communicate is some fashion, an external trigger processor appeared to be long overdue. Hanson [Hans97] started development of the project in 1996.

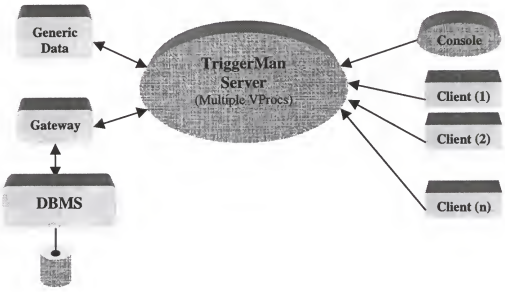


**Figure 5: TriggerMan's software architecture**

The triggering capabilities of TriggerMan provide for both synchronous and asynchronous processing of rules. The benefit of allowing for asynchronous rule condition testing and action execution is reduced transaction duration. The main pitfall of adopting a purely asynchronous architecture is the inability to use triggers within TriggerMan to prevent data integrity violations. Due in part to this limitation, the TriggerMan architecture has been enhanced to enable running natively inside of a database server through the creation of data cartridges or data blades [Info97b]. As the purely external and data blade versions of TriggerMan require the passing of update descriptors in much the same fashion, we discuss only the external version for brevity.

As shown in Figure 5, the basic architecture of TriggerMan consists of the following modules:

1. The TriggerMan server: handles client and console request messages, command parsing, condition matching, and rule action execution.
2. Console: provides high privilege access rights and administrative features
3. Clients: application programs that create triggers, and wait for trigger actions to fire etc.
4. Data sources and gateways: communicate with external data sources and supply TriggerMan with update descriptors. These allow remote access to information maintained in those sources.

TriggerMan's architectural platform, as shown in figure 3, is a shared-nothing SMP-cluster system. The TriggerMan server is comprised of a number of Virtual Processors (VProcs) that can run on different architectural layouts. Each VProc runs on a dedicated processor and contains the following threads:

- A *command server thread* that handles incoming commands and request from client applications or other vprocs.
- A *matching thread* that receives update descriptors and tests them against any related active rules.
- A *rule action execution* thread to execute the matched rules actions.

There are two **API** (*Application Programming Interface*) libraries that allow writing data source applications and client applications. The console and other applications use the client API to connect to the server, send commands, register for events, etc. The data source API allows data sources and gateways to easily send the captured update descriptors to the TriggerMan server.

TriggerMan has been designed to enable the creation of complex rule triggers. The TREAT network and multi-threaded architecture of TriggerMan allow for efficient processing of update descriptors. However, before any matching conditions can be checked, the update descriptors must be detected efficiently, propagated in a timely manner with guaranteed delivery, and the condition action must also be handled in a similar fashion. The ability to efficiently detect, capture, and forward update descriptors both to TriggerMan and to data warehouses is the main focus of the remainder of this dissertation. Without the Data Source Architecture created as a part of TriggerMan, the creation of an asynchronous trigger processor would be impossible.

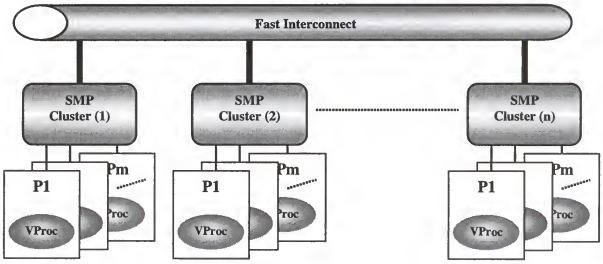


**Figure 6: TriggerMan's architectural platform**

# CHAPTER 3
# TRIGGERMAN'S DATA SOURCE ARCHITECTURE

Before TriggerMan can be utilized to match update descriptors against defined rules, check for temporal conditions, or store information efficiently in a discrimination network, it must first be made aware of data changes that have occurred in selected data sources. A data source is made available to TriggerMan by defining connections, data source types, and schemas for the new source. As the data sources may vary greatly in their data capture capabilities, TriggerMan's data source architecture is designed to be extensible to allow for multiple techniques to be used in data gathering. The TriggerMan Administrator selects the data gathering technique that is best suited for the sites providing data. Future work in the area of choosing the best data gathering algorithm could use sampling techniques to determine data access patterns. Once the data access patterns are known, TriggerMan could be enhanced to automatically adjust the data gathering strategy to the one best suited for the type of data changes occurring in a given data source. Even with automatic adjustments built in, the administrator may have to choose appropriate weighting for CPU, I/O, locking, and message passing to ensure that the adjustments are appropriate for the given business.

## Data Source Definition Language (DDL)

One of the main goals of TriggerMan is to eliminate the need to write applications to handle trigger firing, data sharing, and data manipulation in heterogeneous system environments. To that end, TriggerMan has been developed with a user-friendly, SQL-

like language that can be used to specify simple, complex, and temporal triggers; to specify connection definitions; and to specify data source schemas. The subset of TriggerMan's commands that are used to specify information about connections and data sources are known as TriggerMan's DDL commands.

Borrowing ideas from ODBC and OLE DB [ODBC94], a connection definition must be specified before a data source is activated. Included in the connection specification are the type and location of the data source, the data source name, a username and password, and several other pertinent pieces of information. The general format of the Create Connection command is:

```
create connection <connectionName> (
type=[database|generic],
host="hostname",
dbname="databaseName",
vendor="databaseVendor",
version="databaseVersion",
serverName="vendorSecificServerName",
userid="connectUserID",
password="DbloginPassword",
defaultStatus=[open|closed],
portnum=portNumber )
```

An example of a connection command is as follows:

Example(1): "Create a Connection definition for a Generic Teradata data source".

```
create connection MyConnection (
category=generic,
host="artemis",
dbname="mydb",
vendor="Teradata",
version="1.1",
serverName='TeradataDBServer",
userid="tcc",
passwd="password",
defaultStatus=closed,
portnum=10000 );
```

After a connection has been specified, different sources of data from that connection can be defined. The data source specification is comprised of more than a simple schema definition for a given data source. Although the capability of importing schemas from some connection types is being reviewed, it is not currently implemented. Recall that TriggerMan can receive data from many different types of sources, of which a considerable number will not have a schema definition stored in a catalog that can be queried. In addition, for replicated sources, vertical partitioning capabilities may return a tuple set that does not contain all attributes of the underlying table. Allowing users to input the data source specification enables correct handling in each situation. The general format of the create data source command is:

```
create data source  [connectionName.]sourceName [as localName]
[( [attrName=type [,attrName=type]* ] ) ],
[ with ( [ propertyName=value [, propertyName=value ]* ] ) ]
```

Possible property assignments include:

```
[ type=dataSourceType] [pollingInterval=timeValue] [status= [enabled|disabled] ]
```

Some examples of Data Source commands are as follows:

Example(1): "Create a data source specification for the emp table with the default connection"

```
Create data source emp as MyDataSrc (
empid=integer, age=integer,
lname=varString(20), salary=double );
```

Example(2): "Create a data source specification for the sales table in the connection UF, but do not enable the data source immediately"

```
Create data source UF.sales as ufsales (
acctid=integer, salesmanid=integer,
saleamount=double )
with status=disabled;
```

Basic data source architecture

The basic data source architecture of TriggerMan can be found in Figure 7. The included diagram shows a TriggerMan configuration utilizing the replication capability of a Sybase database.

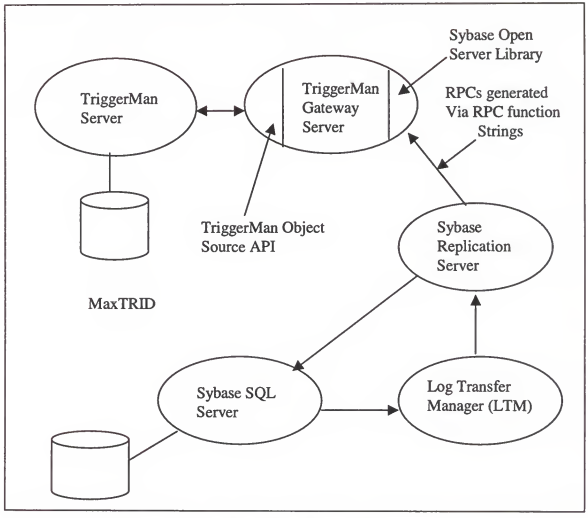


**Figure 7: TriggerMan's data source architecture**

Data gathering in TriggerMan can be subdivided into two broad categories – data sources that forward data to TriggerMan and data sources that require initiative action from TriggerMan. In Figure 7, the former case is displayed. As data is received from a source, the TriggerMan gateway server, or Helper, persistently queues incoming update

descriptors. The temporary queuing mechanism is used to store updates until they can be successfully forwarded to the TriggerMan Server. This persistent queuing adds both functionality and improves the fault tolerance of the system. If the TriggerMan server is temporarily unavailable, the generating source does not need to stop processing until the local queue is full. As time permits, this feature will be extended to allow the DBA to specify a default action of quiescing the source or discontinuing queuing. In addition to improving fault tolerance, the local queuing reduces message traffic by packing multiple update descriptors into a single packet. This feature is extremely important for high-update-rate data sources. If persistent queues are added to the TriggerMan and the Helper processes, rule firing and data sharing across intermittently connected systems can be implemented. At intervals, a dial-up connection can be established to check the queues, pass appropriate update descriptors and update requests, then disconnect.

Data is sent from an application to the Helper process (through RPC's or some similar mechanism provided by the TriggerMan Application Libraries). The Helper process will initially try to forward the information directly to the TriggerMan process that may reside on another machine. To guarantee delivery to a remote site, acknowledgment requests are piggy-backed with the update descriptor. If no response is received, the message is resent. A small queue is used to store messages that have been sent by the Helper until the messages have been acknowledged by TriggerMan. If the message rate from the application is low, this added flexibility will handle short down times of the TriggerMan process/machine. Future design may add persistent storage to the Helper process to allow for extended down times of the TriggerMan server. Sending data to TriggerMan instead of forcing the TriggerMan server to poll data sources

eliminates wasted communication when the Helper process has no data to send. Data is forwarded to the TriggerMan server through message passing to a specific port. Multiple threads are available to the TriggerMan server process to pull updates from the receiving queue to prevent a bottleneck in this area. As the TriggerMan server is designed to run inside of or in association with a data source, the data source can be used to make the receiving queue persistent by changing the update descriptor message to an insert request into a locally stored relation. Pulling threads steadily empty the update descriptor queue or relation and forward the update descriptors to a separate class of threads that compare the update descriptors to the defined triggers. If trigger actions are required, the action is placed upon an outgoing queue where yet another set of threads handles forwarding of the trigger action to the appropriate destination (See Figure 8).

When packets arrive at TriggerMan, they are queued and then consumed by multiple threads within the Asynchronous Trigger Processor (ATP). All rule matching and condition testing is done within the ATP. (One suggested alternative design is to store appropriate rule conditions at each site. If the basic rule is not satisfied, there is no need to forward the message on to TriggerMan. Although this idea has some merit, the drawbacks of double checking rules, extra CPU usage on the client machine, deciding which rules should be stored at which site, and the additional needed memory on each machine has pushed this idea to the background. As the project progresses, this point may be reviewed again).

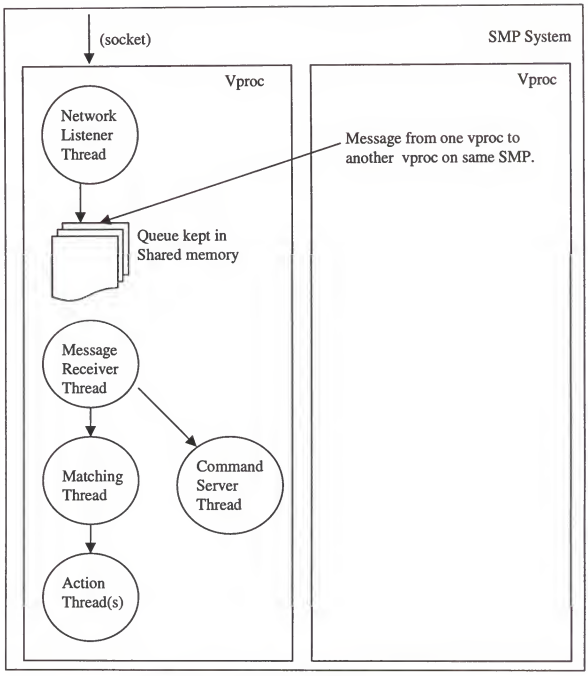


**Figure 8: Data Queuing within TriggerMan**

In addition to communication from data sources, update requests often flow to the data source as a result of a trigger action. In such cases, TriggerMan sends messages to the Helper process with the message once again piggy-backed with an acknowledgment (ACK) request. Here, the Helper process acts like an RPC server in that it queues incoming messages and passes them to the desired application through the provided

library routines. (One modification on this idea is to have one Helper per machine instead of one Helper per data source). This design is similar to the RAISE/REGISTER event mechanisms already found in many systems. TriggerMan uses a Listener to facilitate the raising and registering of events within TriggerMan. The ATP uses a discrimination network to decide what rule conditions have been matched. If a rule condition is matched, the rule action is placed upon the action queue for forwarding to the appropriate Listener or Helper process.

The Helper process provides many benefits to the data source architecture of TriggerMan. The Helper provides message queuing capabilities and acts as an RPC server that is separate from the application. In addition, some limited insulation from inter-machine dependence is achieved by having the Helper and Application reside on the same machine.

As data is passed to TriggerMan, it is immediately stored in queues, and an ACK is returned to the sending process. Threads that are established when the data source is identified at setup time steadily empty the queues. The record is then parsed. If it a command, is processed by the command processor. If it is an update descriptor, a TREAT discrimination network is used to determine rule satisfaction. It is possible that additional information will be needed to determine validity of the rule condition, thus this process will also have the capability of soliciting data sources.

To facilitate communication between processes, and to enable efficient checking of rules, a predefined record layout is necessary. While there has been much work in this area (i.e. CORBA), the design proposed there does not exactly match the requirements of the TriggerMan architecture [Felb96]. The difficulty lies in deciding which data elements

must be included with each type of message. Passing extraneous information introduces excessive network traffic and may require the application to perform additional work before passing data to the Helper. Passing too little information may cause problems by forcing queries back against the database to gather all of the needed data.

TriggerMan must be aware of several categories of information before it can effectively determine if a given update descriptor matches any defined triggers. The first set of information that must be known is comprised of information about the data source itself. TriggerMan must know the name of the data source, characteristics of the data source, including the type and location, and all the attributes and attribute types that comprise the data elements of the data source. This category of information is known as an Augmented Schema Definition (ASD) and a list of the required properties is found in Table 1. The attributes of the ASD do not need to be passed with each update descriptor. Because they are not passed, the ASD is stored within the TriggerMan server catalogs and it is queried each time a record is received. If some types of ASDs prove to be highly volatile, this design may need to be enhanced.

A second category of information about which TriggerMan must be made aware deals not with data about the data source but instead provides details about the change itself. The Update Descriptor (UD) must be able to trace the source of the data, provide information about the type of change that created the update descriptor, as well as chronicle the previous and current value for all attributes that are involved in the change. Additional information may also be required. A list of desired properties is found in Table 2. Depending upon the type of the data source, not all fields outlined in Table 2 may be available for all sources of data. As an example, transaction information for

generic data sources is not needed unless TriggerMan is augmented with transaction handling for flat file systems. However, unlike the ASD, all of the attributes of the update descriptor must be passed for each change to a data source.

**Table 1: Augmented Schema Definition**

| Attr Seq | Attribute Definition |
|---|---|
| 1 | Data Source Name (ie. Application name or DB name) |
| 2 | Data Source Type (ie. Data Stream, Flat File, Relational, Hiarchical) |
| 3 | Data Source Sub-Type (eg. SYBASE, ORACLE, INGRES) (For gateway purposes) |
| 4 | Data Source Sub-Name (ie. Relation Name) |
| 5 | Data Source Sub-Name Attribute Name (ie. Attribute Name) |
| 6 | Data Source Sub-Name Key (ie. Primary Key of Relation) |
| 7 | Data Source Allowed Operations (ie. None, Select, Insert, Update, Delete) |
| 8 | Object Type (ie. char, varchar, int, float4, float8, date, money, image) |
| 9 | Object Length (in bytes; bytes can handle bit representation) |
| 10 | Object Owner |

**Table 2: Update Descriptor Definition**

| Attr Seq | Attribute Definition |
|---|---|
| 1 | Operation Code (ie. Message, Insert, Update, Delete) |
| 2 | Source ID – a surrogate identifier that maps to a Database/Relation Name pair. |
| 3 | Attribute Names |
| 4 | New Object Values |
| 5 | Old Object Values |
| 6 | Forward to All (ie. True if update descriptor is not generated by a request from the ATP, False otherwise. This attribute is used to prevent the circular firing of rules. The Helper is aware of ATP generated transactions so it can prevent repropogation of data to the ATP.) |
| 7 | Transaction Status (ie. B - Begin, C – Complete) |
| 8 | Transaction ID (ie. A unique identifier per transaction) |
| 9 | Transaction Time |

Update descriptors are designed with an internally defined format for improved efficiency. There are several recognized encodings for data record layouts, like ASN.1,

but most introduce a layer of complexity that is not needed for TriggerMan. The record layout for Data Sources is as follows:

1 2 {3} {4} {5} {6 7 8 9}

M M C1 C1 C1 C2 C2 C2 C2

Where M is Mandatory and C is conditional upon the specified element (C1 indicates conditional upon element 1). Operation codes can be abbreviated to reduce record size. The meanings of 1 through 9 are found in Table 2. Attribute 7 of the tuple descriptor format is not intuitive in its usefulness; thus we address it further here.

A significant drawback of uncoupling a trigger condition from the issuing transaction is the possible change of database state between the creation of the update descriptor and the time the trigger condition is checked. If TriggerMan utilizes all virtual nodes in the TREAT network, the originating data source must be queried again to determine whether certain types of triggers are valid. In such a scenario, the database state could change and a trigger that may fire using a coupled approach may not fire using the uncoupled design.

If the TREAT network utilizes stored alpha and beta nodes, a consistent snapshot of the state of the old database can be maintained. The issue associated with this approach is ensuring that rules are not tested until all parts of the transaction that created the update descriptor are applied to the corresponding older snapshot. The deferring of the trigger condition testing to the end of the transaction is standard practice in most of the commercially available database products. Although immediate checking is logically sound for many types of triggers, both join and aggregate triggers could yield errant results if trigger condition testing is performed before all changes for a single transaction

are applied to the snapshots that are held within TriggerMan. It is for this scenario that attribute 7 was introduced.

When using a change detection technique that provides some information about the issuing transaction, attribute 7 is used to ensure all update descriptors associated with the transaction are applied to the replicate snapshot before any rule condition testing is implemented. Not all change detection techniques and data sources are able to communicate transactional information, however, thus immediate testing in these cases is the only viable alternative.

One last feature of the TriggerMan architecture that is not yet completely specified but that can lead to a great improvement in performance is the pushing of some condition predicates to the Helper process. The Helper process will typically be installed on the same machine as the data source. It can eliminate much unnecessary network traffic if it is effectively used to provide filtering for the data source.

An aspect of the data source architecture not pictured in Figure 7 is the need for the Gateway Server to be able to send requests back to some connection types. This issue will be addressed in greater detail in section on Queriable Data Sources.

### Data Stream Data Sources

Data stream data sources (DSDS) are essentially insert-only data sources. This type of data source is quite common in the manufacturing environment where sampling data from the manufacturing floor is forwarded to a database. A minicomputer (i.e. PDP11) or PC system that forwards the information to the production database typically performs the polling. RapidBase is a commercially available main-memory database

product that has been specifically designed for use in just such an environment [Rapi96][Wols96].

The data stream data source is not queriable, nor does it have persistent state, so update descriptors must be passed to TriggerMan. The rules within TriggerMan can be used to check the update descriptor for boundary conditions before inserting the data into a persistent database. If a particular sample fails to meet some criteria, TriggerMan can insert the sample data into an error table or send a notification to the shop floor.

If a data stream data source application wishes to communicate with TriggerMan, it must first link to the provided TriggerMan library. The libraries can then be used in one of two ways. If persistence is required, the data is forwarded to the TriggerMan Helper process that is typically located on the same machine as the polling process. The Helper process then forwards the information on to the Server. For faster response when either only one machine is being utilized or persistence is not required, the TriggerMan Library enables direct communication to the Server through RPC calls. The basic process layout for a generic data source can be found in Figure 9.

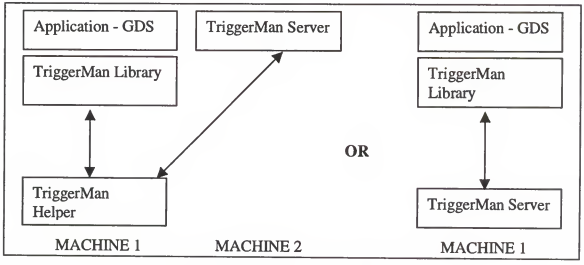


**Figure 9: Basic Process Layout for GDS**

Replication Based Data Sources

Some connection types such as Sybase, Oracle, Informix, and INGRES have replication capabilities built directly into their core products. Figure 7 shows the basic data flow and process organization for using TriggerMan with replication-enhanced systems. The replication capability of the DBMS can be used to forward pertinent update descriptors to TriggerMan without greatly impacting the performance of the underlying database. This type of architecture shows a great deal of promise for reducing the amount of network traffic by minimizing the passing of extraneous update descriptors.

The replication process can often be tuned to return horizontal and vertical partitions of the data. The schema and domain of the forwarded partitions can be modified based upon the triggers that have been defined for TriggerMan. Using partial instead of full replication will necessitate the modification of replication rules whenever triggers are defined that require information outside of the previously defined restricted view. Automating the replication setup process for each system may prove to be difficult. One possible problem of replicating partitions instead of replicating entire relations is the inability to maintain a complete copy of the relation within TriggerMan. In some scenarios, it will be more cost effective to replicate the entire table to reduce TriggerMan's need to query the data source to gather the information necessary to determine the satisfaction of complex rule conditions.

Replication servers typically place update descriptors into a queue or into a temporary database until all subscribers have been notified of the change. A different Gateway Server must be implemented for each type of replication based data source to

enable the handling of different formats and different publishing scenarios. Each Generic Data Source (GDS) utilizes the same Data Source Application Program Interface (DS API) to pack the data and forward the information to the TriggerMan Server.

Trigger Based Data Sources

Identifying modified data sets without scanning the log file can cause performance degradation in the system. One way to reduce extraneous costs as much as possible is to use the triggering mechanism provided in many database products. As new data sources are defined within TriggerMan, a simple trigger can be created within the supplying database to identify when changes occur to the requested resource. A benefit of this architecture is that the trigger can supply both the modified tuple and old state of the tuple. Passing of an old/new value pair eliminates the necessity of storing local state within TriggerMan. A second benefit is the possibility of using rules that are triggered on Select operations to notify TriggerMan of access attempts.

Using a few simple rules to identify when basic changes occur within a data source does not greatly hinder application performance. Generally, the performance penalties of using triggers are caused by the action of the trigger, if only a few triggers are defined, and not the checking of the trigger. As all current products use a direct coupling mode, the action of the trigger is performed as a part of the underlying base transaction. Thus a simple insert into a single table could lead to an update being issued against ten or more additional relations. The original insert cannot commit until all corresponding updates have completed. In addition to the action slowdown, defining more than a few triggers against a single relation can also yield a performance penalty. Most major

relation products, such as Informix, will not allow you to define more than thirty triggers against a single relation.

Triggers that are used to identify updates to a data set can be utilized in several ways to pass information to TriggerMan. If the triggering database supports internal procedures with RPC capability, the procedure can issue a simple call to pass the information to the Helper process. The RPC can send updates directly to TriggerMan for high throughput systems if the possible loss of update descriptors can be tolerated in cases where the TriggerMan server is temporarily unavailable. This is an efficient solution, but it is not a robust one. Recall that currently all procedures are defined to be issued as part of the transaction. If this is the case, a rollback issued after the firing of the RPC will artificially signal an update to TriggerMan. An improvement on this approach is to simply use the RPC to notify TriggerMan of a possible change. TriggerMan is then responsible to query the source to ensure that the update actually did occur.

If TriggerMan is going to be required to query the database, another solution is to have the internal trigger propagate updates to an update queue table. Propagation requires that TriggerMan periodically query the update queue table to determine what changes have occurred. After the data is safely transmitted to the persistent queue within TriggerMan, the Helper process removes all transmitted records from the update queue table. This approach allows only committed updates to be forwarded to TriggerMan. The pitfall is the overhead of artificially lengthening the issuing transaction. For every table X in transaction T that has a TriggerMan rule defined against it, the cost of modifying the table is essentially doubled by this approach.

The use of triggers to identify and forward modified tuples to a separate table has several variations. A referenced table can have one update queue table, or separate ones for insert, update, delete, and possibly select. To reduce the number of tables, one table can be used to gather updates for all relevant tables. Yet another solution is to create one table that is used to store only old and new values of the modified attribute(s), not tuples. Each row would also contain a relation name and attribute name. The table schema must have an old/new value pair for each attribute type supported by the system or it must have a column indicating the native type and a conversion function that can be used to return the stored data to the appropriate format. The high access rate of this table, one insert for each attribute of a table on which an insert was performed, can make the cost of this solution prohibitive. Each of these variants suffers from the same artificial lengthening of the initial transaction.

Differencing Based Data Sources

Many types of data sources will fall into this category because it offers promise for identifying modifications regardless of the underlying capabilities of the data source. Even if the underlying source has triggering or replication capabilities, a differencing approach can be selected. The subtle point of this technique is the necessity of determining both that a tuple has changed and determining the exact nature of the change. The performance characteristics of previous pure differencing techniques were poor, especially with large data sets. Several new algorithms have been developed to improve performance using a modified differencing technique. The new differencing algorithms are discussed in chapter 4 and their performance characteristics are shown in chapter 5.

Local Data Sources

The multi-threaded, queuing architecture of TriggerMan enables rule condition processing in a high transaction environment. If a data source is local to TriggerMan and performance is critical, TriggerMan provides an additional interface to reduce the need for an intermediate Helper process. The local data source can use RPCs to directly communicate update descriptors to TriggerMan. This separate interface eliminates the relatively high cost of message passing between the source and TriggerMan.

In some environments, like during the manufacture of fiber optic cable, the sampling occurs at a higher rate than can be written to the database. The information is typically queued in a machine with a large amount of memory until the database writes can catch up with the polling process. The "catch up" is made possible due to the non-constant nature of the cable manufacturing process. In such an environment, the local data source interface is necessary to enable the processing of update descriptors arriving at such a high rate.

Application Based Data Sources

TriggerMan provides many different approaches for gathering update descriptors from the actual data source. If data is gathered directly from the source, no application changes are necessary. Another technique to gather information is to modify the application to pass updates to TriggerMan. Front-end passing of modifications can be separated into two broad categories – Application Modification and Middleware Modification.

Application Modification

An application can be modified to send update descriptors directly to the Helper process by linking with the data source API library. After an application issues an update and validates the return status, it can forward the update descriptor to TriggerMan by calling the appropriate Library function. For applications under development, this functionality may prove to be cost effective. Traditional applications often perform selects before issuing updates. This read enables the sending of state information with an old/new value pair as well as simple modify notifications.

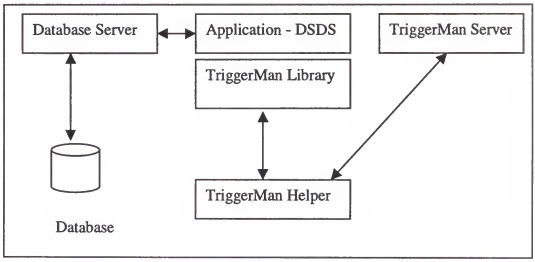


**Figure 10: Application Modification Using Data Source API**

Middleware Modification

Another approach to discerning updates from the application without actually modifying old code is to enable the ODBC driver to pass information to TriggerMan. Several vendors have developed additional layers that can also be implemented on top of currently available ODBC drivers [USOF96]. Enabling any of these drivers to forward changes to TriggerMan does not appear to be difficult (see Figure 3). As update

descriptors are passed to the ODBC driver, a TriggerMan procedure can be used to store the update request. ODBC is required to accept a return status to determine if the update was successful. A second TriggerMan procedure can be used to trap this response to ensure that only committed update descriptors are forwarded to TriggerMan. Assuming a positive acknowledgement from the database, a third TriggerMan procedure can be utilized to forward the results to the Helper process. With the middleware approach, only the registering of the update descriptor increases the initial transaction length.

Queriable vs. Non-queriable Sources

A queriable data source is one that provides an interface that can be used to gather information from the source. An interface typically receives a request and returns a data set. With the improvements to ODBC, the number of sources providing an SQL-Like query interface has escalated to more than 130. The introduction of wrappers shows some promise in quickly increasing the number and type of queriable sources. The cost of gathering data from a data source that provides such an interface is typically less than the cost of gathering from non-queriable sources. The differencing approaches that will be discussed in chapter 4 outline some of the reasons for the reduced cost.

For non-queriable data sources, the need for application based approaches is more clearly evident. A source may have query capabilities that are not made available to TriggerMan. For other sources, the queriable interface may not have been defined. Additional sources may be data-stream data sources with no persistent state. Each non-queriable type provides its own set of challenges.

Recall the two-fold need of identifying that a change has occurred and recognizing the nature of the change. Probably the most efficient approach for identifying that a change has occurred in a source with persistent state is to use a modified ODBC or application interface to trap for update descriptors. As the update descriptors are captured, they can be passed directly to TriggerMan or the Helper process. This approach removes the need to copy entire data sets to TriggerMan before comparisons.

Once an update descriptor has been identified, it must be compared against the old state before being applied to that state. TriggerMan can be used to store the old state thus enabling the creation of a larger set of meaningful triggers. In addition, join triggers to the non-queriable source can be implemented due to the state of the non-queriable data source being stored within TriggerMan.

Transactionally-Consistent Update Propagation

Just as important as the gathering of information is the propagation of data to selected sources. After TriggerMan gathers data and checks rule conditions, it must often forward or initiate changes to other sources. Ensuring that change requests are persistent and that they are applied in a transactionally consistent manner is an important aspect of the TriggerMan architecture. If a large number of updates are requested, it is also critical to apply the changes in a manner that minimally affects performance on the receiving data source (i.e. data warehouse or decision support system) [Kawa97].

Update requests from TriggerMan to sources can be broken into three broad categories – single-statement requests, multi-statement requests to a single site, and multi-statement requests involving more than one site. The basic architecture to ensure

the persistence of update requests is the same for each category. Multi-statement heterogeneous update requests necessitate the building of additional functionality into TriggerMan. The use of queues to facilitate such functionality has been well known since the work by Bernstein [Bern90][Bern90b][Bern91].

The action of a TriggerMan trigger can request that an update occur at the originating or some additional site. This request is not acted upon immediately, but is instead placed upon a persistent outgoing queue within the TriggerMan server. The update descriptor is tagged with an internally assigned transaction id, with a total indicating the number of statements within the transaction, and with a request number indicating the order of the request within the transaction. The update descriptor on the queue must also contain information about the connection types and addresses of the connections that contain the relations referenced in the trigger action. If the information is not directly stored with the update descriptor, the outgoing thread must look up the connection information, which is stored in TriggerMan catalogs, before forwarding the request to the Helper process of the connection. The request set is left in the TriggerMan persistent queue until the Helper process acknowledges receipt of all parts of the transaction. The Helper process, after examining the tagged fields, returns an acknowledgement so that the TriggerMan process can remove the requests from the outgoing queue found within the server.

The Helper process reads requests from the Helper persistent queue and applies them in a transactionally consistent manner. The Helper uses the library functions or an ODBC interface to submit the requests to the data source. The Helper checks for errors after each statement. If all statements issued by the Helper return with a valid status, a

commit is forwarded to the data source. If a commit acknowledgement is returned to the Helper, the Helper removes the entire request from the queue. If no reply is received, the Helper process may have a difficult time determining if the acknowledgment was lost after a successful commit, if the commit request was lost, or if the system is simply extremely slow. If a second update were to be successfully issued, TriggerMan could not be sure if the data source had been down and come back on line, thus the commit request committed an empty transaction, or if the second commit actually committed the original multi-statement transaction. To decrease the likelihood of such a scenario, each originating source can be augmented with a table to contain transaction Ids for TriggerMan transactions. The protocol for updates from a Helper is to first insert the transaction identifier into the local source table as part of the transaction. If the Helper does not receive a commit acknowledgement, the Helper queries the Helper Table to determine the existence of the transaction id. If the system failed before the commit, the transaction id will not appear, and the Helper can reissue the transaction. If the ID is present, another commit can be issued. If the transaction is already committed, the commit of a blank transaction does no harm. This second commit insures that the original commit was not lost. This can repeat until an acknowledgement is received. After a successful commit, the transaction can be removed from the Helper queue. On a multi-processor system, a single row table to hold the transaction id will not suffice, but this can be easily extended to a multi-row Helper table. After the status of the commit is checked, the Helper removes old requests from the Helper persistent queue.

Although this protocol is sound, the actual need to forward commits multiple times is quite low due to the Helper process residing on the local machine. The removal

of multi-site message passing from the transaction semantics greatly improves the likelihood of single-pass success.

For multi-statement heterogeneous updates originating from TriggerMan, the locking protocols outlined by Skeen [Skee81] are not sufficient. This area needs further investigation to optimize performance of multi-statement heterogeneous transactions. Some promising protocols include modular and decentralized three phase commit [Guer96a], decentralized non-blocking atomic commit [Guer96b][Guer95a], and dynamic terminating multi-cast[Guer95b].

If a trigger is defined that requests a distributed, heterogeneous update, the update requests can be grouped into a single transaction, or options can be selected to handle each action or set of actions at a single site as separate transactions. If each part of the action is handled independently, the queuing mechanisms outlined previously are sufficient. If it is necessary to treat the multiple update requests as a single transaction, Bea Tuxedo [BeaT99] and Encina [Enci98] are two commercially available queuing products that can be used to facilitate distributed, heterogeneous transactions. The unnecessary use of distributed transactions should be avoided as the likelihood of a single failure at a given point in time is higher with multiple systems than with one. It is unadvisable for locks to be held on all relations involved in a transaction while waiting for a single site to become available again. If the desire is to apply changes in the order in which the issuing trigger fired, then subsequent single site requests must be deferred until the distributed transaction can complete. Given this scenario, the failure of one site can prevent updates from being applied at all other sites. Thus, the protocol must be altered to allow for the out-of-order application of trigger actions or distributed transactions

should be avoided whenever possible. Because TriggerMan does not support the notion of a global state, the requirement to use truly distributed, heterogeneous transactions are minimized. The delivery and application of the update requests to the site that is unavailable is guaranteed due to the internal queuing mechanisms that are built into the Helper and Server processes. Figure 11 outlines the communication steps involved in a typical distributed transaction that is triggered by an insert into an Informix database. The trigger definition is found in figure 2.

***Round 1*** – An insert is performed against a relation in an Informix database

***Round 2*** – Simple triggers within Informix cause an update descriptor to be inserted into the server's persistent update descriptor queue.

***Round 3*** – Matching threads pull update descriptor from persistent queue. After checking the update descriptor against defined triggers, the matching trigger actions are inserted into a persistent outgoing server action queue. The update descriptor is then deleted from the update descriptor queue.

***Round 4*** – Copy action request to persistent queue at applicable helper locations. If the delivery is successful, remove the request from server action queue. If the delivery is unsuccessful, maintain the action request in the server action queue and re-attempt delivery at either timed intervals, receipt of a wake-up acknowledgment from helper, or upon a user request.

***Round 5*** – Simultaneously, all helpers apply single site trigger actions to the corresponding data sources. If any site is unavailable, that sites request remains queued until successful. Other sites complete if possible.

***Round 6*** – After helper receives successful completion response, the update request is removed from the helper request queue.

**Figure 11: Typical Distributed Transaction Communication Architecture**

# CHAPTER 4
# DIFFERENCING-BASED CHANGE DETECTION

An alternative approach to change detection, proposed by Chawathe, [Chaw96] [Chaw97a][Chaw97b][Labi96] suggests comparing two versions of the data set to identify change. This technique, known as *differencing*, can be used with most data sources, including both structured and semi-structured data. Although differencing is not traditionally recognized as an optimal solution for isolating changes in data sets, it does provide a reasonable option for environments where more efficient mechanisms, such as triggers and replication, are not available [Lind86][Labi96]. The main pitfall of differencing is the relatively high cost of each relation comparison. Previous attempts at reducing differencing costs on structured data use sorting and compression techniques at the local site to lessen the I/O required to compare multiple copies [Chaw97b][Labi96]. Minimizing the costs of gathering changes in semi-structured data is accomplished by using lowest-cost edge detection algorithms where an edge is the step necessary to transform one aspect of the old image to the new image [Chaw97a][Chaw96]. The techniques used in both instances do not yield sufficient performance for commercial applications due to I/O and locking constraints, as will be demonstrated in chapter 5.

The basic tenant of differencing is to use some type of difference command, like the Unix or VMS "diff," to determine the difference between an older copy and the current data set. Given a data set of size $n$ pages, the space complexity for a simple differencing system is at least $2n$, $n$ pages for both the original and the replicate. As the

algorithm must compare corresponding rows between the two data sets, the I/O complexity required to identify all of the data changes can be as high as $n^2$. If insufficient memory exists to contain $n+1$ pages, each page reference of n can require an entire table scan of the replica, thus yielding $n^2$ I/Os. A poor implementation of this algorithm, in which each tuple requires a scan of the replica, can lead to even greater I/O costs.

Previous differencing algorithms seem to have overlooked the fact that change detection is a two-stage process. First, the actuality that something in the set (tuple, data set, or image) has changed must be recognized. Second, the nature of the change must then be isolated. The existence of a change implies that the data source is actively being modified. The update rate and the frequency at which the differencing algorithm is run greatly affects the expected number of changes detected at each iteration of the algorithm. Earlier estimates of a 20% change in data between comparisons [Labi96] are valid for relatively few business applications, even if the algorithm is run once a day. A more realistic estimate is a 1% change per day for the majority of business related applications.

If differencing is deferred until the system is quiet, perhaps the previous algorithms proposed by Chawathe and Labio are sufficient. Not all change detection can be postponed until the system is inactive, however. If modifications must be detected shortly after they occur (i.e. for use with an asynchronous trigger processor [Hans97]), the basic differencing algorithm is not adequate, especially for relatively large data sets. In addition, writing differencing applications for each source of data is not sufficiently cost effective to make it a viable solution.

In an effort to improve concurrency, reduce I/O, lessen space complexity, and eliminate the need to write a differencing application for each data source, several new

algorithms are introduced here. These algorithms are designed to work with relational and object-relational database data sources. As a practical matter, they can be implemented for any ODBC-compliant [ODBC94] data source of which there are more than 130 currently available. The introduction of *several* query-based techniques instead of only one enables full utilization of the capabilities of each data source.

ODBC compliant data sources generally fall into three broad categories – those having modifiable table schema definitions, those with modifiable database schema definitions that disallow changes to existing table schemas (i.e. new tables can be created), and those that are queriable where neither the table nor the database schema is modifiable. If the source data repository can perform joins *and* allows a duplicate copy of the data to be maintained at the same site, the source data server can be used to detect appropriate changes. If either of these conditions fail, then the original, or some part of the original, must be forwarded to a separate remote site to maintain the copy. It is assumed that the remote site can perform joins. In both the single and remote site scenarios, the following algorithms represent several query-based approaches that can be easily implemented to create realistic differencing applications against each category of data source. The basic algorithms are outlined below. These algorithms are further explained in later sections.

- **Modified Table with Time-stamps (TS)** – For use with systems where the underlying "relation" can be augmented with a time-stamp attribute. This algorithm yields good performance for insert and update-only systems. It is especially applicable for systems that can automatically maintain the time-stamp attribute without requiring application modification.

- **Modified Table with Time-Stamp and Delete Fields with Non-clustered (TDI) or Augmented Non-clustered Index (ATDI)** – For use with systems where the application code and the base relation can be modified. Both TDI and ATDI introduce a secondary index to eliminate the need to scan the base relation to identify changed tuples, thus reducing the number of records that must be locked. Augmenting the secondary index with appropriate attributes (ATDI) eliminates the need to join back to the base relation and can be proven to yield optimal I/O cost if the data source supports compression. This algorithm quickly identifies changes with minimal I/O and does not appreciably lengthen on-line transactions. The application code must be modified to make use of the new time-stamp and delete attributes.

- **Modified Table with Tuple Level Checksum (TLCS)** – For use with systems where either the table schema or the database schema can be modified but the application code cannot be changed. Instead of the use of a time-stamp attribute to help identify which tuples have been modified, a checksum is calculated and maintained with each tuple. A separate process scans each relation, comparing the tuple to the checksum. If a discrepancy is identified, the tuple is flagged as having been modified. This approach does not perform well against large systems that allow *deletes* as a full sort merge join with outer join detection must be used to identify deleted rows, thus negating the benefit of the checksum.

- **Modified Database Schema (MDS)** – For use with data sources where the database schema can be modified but the relation schema cannot be changed. To identify changes, a full outer-join of the two relations (old and current copy) must be

performed. Excellent performance can be realized if the relevant snapshot, the subset of attributes and tuples, is highly restrictive. The worst-case scenario of this technique yields the same performance characteristics of the best-case previous algorithms. This technique can induce locking problems for large, non-restrictive snapshots, as the two relations must be scanned.

- **Local Site Incremental Differencing (ID)** - For use with extremely large data sets where the database schema can be modified and sufficient space is available on the local system. Similar in technique to MDS, except that ID eliminates locking contention through incrementally differencing subsets instead of comparing the entire relation. Although I/O is comparable to the best-case older algorithms, this technique does not suffer from a serious lock contention problem.

- **Remote Site Incremental Differencing with Range Checksum (ID/RCS)** – For use with systems where the local site cannot be modified or insufficient space is available at the local node. This algorithm combines a variant of the techniques used in TLCS, MDS, and ID. An incremental join is performed against the old and new copy, where the new copy is relocated to a remote site. To reduce the number of tuples of the relation at the source site that must be copied to the remote site before comparison, a checksum is assigned to each range of data of the source site instead of to each tuple. The range checksum is maintained at the remote site and is used with the query of the source site to determine if a change has occurred somewhere within a given range. The incremental nature of this algorithm eliminates the locking constraints of other algorithms. An example of this technique is found in section 4.6.

- **View-based Differencing (VBD)** – For use with systems where snapshots are based on a particular view, as in data warehouses. Variants of this technique can be used to greatly improve refresh costs of the data warehouse view by isolating changes at the view level instead of the relation level.

As each algorithm is most appropriately used with a given class or category of data source, the algorithms are separated in later sections by the distinguishing characteristic of the data source – query only, modifiable database schema, and modifiable table schema. To ensure a better understanding of the usefulness of each algorithm, a brief description of the key issues related to differencing-based change detection precedes the algorithm descriptions.

Differencing Issues in OLTP Systems

Determining changes in OLTP systems necessitates addressing concurrency issues. In almost all structured data systems, a read-lock is imposed on a tuple or page when data is read into memory. When scanning tuples in a system where data is stored as relations, locks must typically escalate from page or row level to a table level lock if roughly more than ten pages are accessed [Ingr91]. (This is often tunable, but here we are discussing the general case.) The loss of concurrency caused by the acquisition of a large number of locks during a differencing scan can prohibit a differencing technique from being utilized on a frequent basis if locking is not properly addressed. Frequent change detection is critical for both Asynchronous Trigger Processors (ATP) and systems where the amount of difference between two replicate snapshots is restricted.

Many algorithms, like sliding-window and pre-sorted sort-merge [Ull89][Labi96], have been introduced for sorting then comparing a current copy to an older one. In each, the cost of every differencing scan against a data source of size $n$ pages is $n^2$ for semi-structured data and $2n\log n + n + m$ for structured data. The cost to sort the base relation is 2nlogn and $m$ is the size in pages of the duplicate copy. As the size of the duplicate and the base relation are essentially the same for these algorithms, the cost can be approximated to $2n+2n\log n$ if the copy is already stored in sorted order. This cost model is correct assuming memory is not sufficient to contain $2n$ pages and that the algorithm always uses a sort merge join with outer join detection. With enough memory (lMemoryl > Sqrt($n$)), the costs can be reduced to approximately $6n$ [Labi96]. The cost of the comparison can be reduced to $2n$ if a sliding window comparison or a pre-sorted sort-merge technique is used [Labi96]. The sliding window protocol requires the strict assumption that the data cannot be restructured (e.g. modified from a B-tree structure to a hash structure). Pre-sorting techniques require that the relations are stored in sorted order. Requiring static or pre-sorted structure invalidates this option for many data sources.

To force sorted order, a query can be run to create a separate file that is used in the comparison step. Requiring the creation of yet another copy increases the comparison costs of [Labi96] to $8n$ and increases the space complexity to $3n$. If the underlying relation is not sorted, the comparison cost for large tables is prohibitive. For example, sorting a 200,000-tuple table can take several minutes and will acquire a read-lock on the entire data set. Locking the entire data set during report creation is generally not acceptable for OLTP systems. Removing read-locks from the scan of the base relation

improves concurrency but it allows for the introduction of dirty reads that can corrupt the copy of the base relation.

Current update inference algorithms [Labi96] ignore or do not sufficiently address issues such as locking and excessive I/O. In addition, their applicability for general use is questionable given their inability to determine differences between two sources that are not simply flat files. The existence of escape sequences and format characters in many data sources prevent the use of a simple "diff" command. Even in situations where locking is not critical and the files being compared are flat files, the cost, $2n$, of the best case algorithm proposed by Labio [Labi96] can still be prohibitively high if the difference operation must be run more frequently than once a day. In this paper we outline several new algorithms that are applicable for most structured sources of data (i.e. those that are queriable through a gateway such as ODBC.) The performance characteristics of each algorithm are given along with descriptions of expected locking constraints. These new techniques make use of meta-data and recent technology advancements in heterogeneous connectivity to achieve excellent performance.

## Query-Based Differencing

The concept of query-based differencing is much the same as the techniques outlined by Labio, Chawathe, Garcia-Molina, and Widom in that two copies are compared to identify some type of data set change. Query-based differencing is different, however, in three main areas. First, the comparison is performed through the use of a query language, such as SQL, so the benefit of data-independence, projection, and selection can be utilized in the comparison step. Second, the copy is not required to be an exact duplicate of the original as only pertinent subsets of data are of interest. Finally,

pure comparison to identify changes can be augmented with flags and time-stamps, much like [Lind86], to reduce the cost of identifying which tuples have been modified.

Query-based differencing can be accomplished by using SQL with a "gateway" such as ODBC to gather differences from heterogeneous data sources [Geig95]. The large number of ODBC drivers currently available enable relatively seamless access to both structured and semi-structured data sources. The drivers understand a specific syntax of SQL, translate the query to the native SQL of the underlying data source, if applicable, and then either directly or through the native data server gather results from the data source [ODBC94]. The gateway may need to reformat the return values (e.g. date fields) before passing the results back to the front-end application, but the SQL application does not require modification even if the data source is converted to a different DBMS product. Often ODBC drivers for simple data sources are enhanced with database server capabilities to provide both join and optimization functionality.

The query-based differencing approach provides several benefits over simple flat file differencing. As an example, most data sources that contain information of interest are not simply flat files. A UNIX "diff" command cannot be run against two EXCEL or LOTUS spreadsheets because the underlying files contain many special characters that make the comparison difficult. Exporting data to a flat file before analysis can simplify testing but it is inefficient and completely unnecessary given recent advances in ODBC technology. As such systems are ODBC compliant, the appropriate driver can be utilized to perform an SQL join operation of the original data set with an older copy (or one of several other techniques listed in later sections) to determine the differences. An SQL-based technique does not require the pre-sorted or static structures indicated in previous

algorithms but instead uses the optimizer of the data source to determine the best search strategy. The optimizer often has knowledge of selectivity factors and indexes that can be used to help reduce the cost of change detection. The use of a local server can further reduce the cost of subsequent comparisons as previously cached data may remain in the server. In addition, the appropriate use of SQL allows for data independence so that schema modifications do not "break" the differencing program.

**Table 3: Definition of Terms**

| N | Definition |
|---|---|
| A | Asynchronous Trigger Processor |
| T | Time-stamp attribute |
| D | Delete Flag attribute – one bit or byte depending upon |
| $B$ | Base relation used for comparisons |
| $B_a$ | New B augmented with TS&DF |
| $C$ | Copy of B excluding changes occurring since the last difference |
| $C_r$ | Copy of B restricted to pertinent attributes and tuples |
| $X$ | Number of Tuples in $B$ |
| $Y$ | Number of Tuples in $C_r$ |
| $T$ | Time-stamp with delete flag algorithm with secondary index |
| $A$ | Time-stamp with delete flag algorithm with augmented secondary |
| $T$ | Modified table with Tuple Level Checksum algorithm |
| $M$ | Modified Database Schema algorithm |
| $ID$ | Local site incremental differencing algorithm |
| $ID$ | Incremental differencing with range-value checksum algorithm |
| $V$ | View-based differencing |

Not all ODBC compliant data sources provide the same functionality, however. A data source may allow changes to relation schemas, the database schema, or it may disallow any type of change. In order to take full advantage of the capabilities of each data source, the query-based approach provides a suite of algorithms that can be selectively used in different environments. Tables 3 and 4 outline the variables that are used to evaluate each algorithm.

**Table 4: Definition of Variables with Default Values**

| *V* | Definition | Default Value |
|---|---|---|
| *S* | Size of $B_a$ in Pages. $S = X/(P/(T+A))$. It is possible to increase the tuple size without increasing table size if $mod(P,T)>(P/T)*A$ | 1.1 N |
| *M* | Size of $C_r$ in pages = Y/(P/(size(key)+F)) | .3 N |
| *K* | Size of Key in Bytes | 8 bytes |
| *F* | Sum of the size of applicable fields requiring change | |
| *N* | Size of *B* in Pages | 1000 pages |
| *P* | Page Size in Bytes | 2048 bytes |
| *T* | Tuple Size in Bytes | 200 bytes |
| *A* | Size of TS+DF in bytes | 13 bytes |
| *I* | Size of Secondary Index on *B* in Pages, ie. | .1N |
| *R* | % of rows applicable for change detection | 100% |
| *Δ* | Percentage of table to change between scans | 1% |
| *Z* | Number of relations involved in VBD join | 4 |
| *K* | Constant – height of index | 3 |

Modified Database-Schema Approach (MDS)

If the local site can be augmented with a pseudo-duplicate copy of the base relation, no change to the original table or application programs is needed. If the local site data server has a moderately sized cache, and the difference operation is performed against small tables at fairly frequent intervals, it is possible for both the original and the duplicate relation to remain cached in local server memory. In this case, the cost of performing subsequent joins is 0 IO and minimal CPU usage, which is better than the *2N* IO required for each invocation in the best case snapshot scheme as the previous techniques must read data from a file before comparing. Figure 12 shows the basic algorithm for using SQL to determine the differences between the original – *basetable* with attributes *key, t1, t2,…,tn* – and the copy – *duplicate* with attributes *key, t1, t2,…,tm.* Note that the duplicate may be a subset or superset of the base relation if only certain attributes have relevance to the desired snapshot. SQL-92 simplifies this query with the full outer-join operator, but the algorithm as stated is designed to be as generic as possible and as such uses UNION instead of the SQL-92 OUTER-JOIN operator.

```
[1] Select change_type='Update',t1.key,t1.f1,t1.f2,…,t1.f_m from base-table t1,
    duplicate t2
[2] Where t1.key = t2.key
[3] And (t1.f_a!=t2.f_w or t1.f_b!=t2.f_x or t1.f_c!=t2.f_y or … or t1.f_n!=t2.f_m)
[4] {And Restriction(t1)}
[5] Union
[6] Select change_type='Insert', t1.key,t1.f1,t1.f2,…,t1.f_m from base-table t1
[7] Where not exists (select t2.key from duplicate t2 where t2.key = t1.key)
[8] {And Restriction(t1)}
[9] Union
[10]   Select change_type='Delete', t2.key,t2.f1,t2.f2,…,t2.f_m from duplicate t2
[11]   Where not exists (select t1.key from base-table t1 where t1.key = t2.key)
[12]   {And Restriction(t1)}
```

**Figure 12: Simple Query-Based Differencing Algorithm (QDA)**

By definition, a full outer join is comprised of three component – a join, a left outer join, and a right outer join. In the QDA algorithm, lines 1 through 4 perform a join between the sets utilizing the primary key of each relation. Tuples containing a key that does not reside in the intersection of the two sets do not appear in the newly formed join set. Lines 6-8 simulate a left-outer-join by identifying all tuples with a key in the base-table that is not found in the duplicate by using the not exists operation. Finally a right-outer-join is simulated in lines 10-12 to find all deleted tuples by finding all keys within the duplicate that are not found in the base-table. The clauses in lines 3, 4, 7, 8, 11, and 12 can further restrict comparisons to subsets of the base sets, thus improving performance. Line 3 is utilized to determine changes to specific attributes within a tuple.

The SQL algorithm restricts comparisons to the pertinent attributes and tuples of the snapshot by modifying the fields referenced in the SELECT and WHERE clauses. In addition, if relevant changes are restricted to a subset of the tuples, the WHERE clause can be modified to select only appropriate rows from the original relation. As an example, *Restriction (t1)* above could be *t1.dept='ABC'*. The algorithm can be easily generalized to

work with any SQL-based system. If the number of updates is high in comparison to the ones that apply to the snapshot, these capabilities can greatly improve performance by reducing the number of spurious messages forwarded by the detection algorithm. The reduction in the number of unnecessary messages can be quite significant.

As an example, an externally defined trigger or a replicated copy (snapshot) may only be interested in *salary* changes (size *8* bytes) to persons in department *ABC*. The *Personnel* table, *P*, is comprised of *21* attributes with a total tuple size of *200* bytes and a key size of *8* bytes. On average, approximately *10* records fit per page. Assuming *10,000* employees, the size of P, designated *N*, will be *≈1000* pages (ignoring index overhead). Also, given the assumption of equal distribution between *100* departments, there will also be *≈100* persons per department. Finally, assume field changes are equally distributed between all non-primary-key attributes.

With the pre-sorted and sliding-window algorithms, the cost of comparison, even assuming presorted data, is at least *2N* and is more likely *8N* as *P* cannot be required to be sorted for most systems and flat files are used during comparison. In addition, of the number of detected changes, very few are actually applicable (only those pertaining to dept ABC). In this example, the number of applicable changes detected is *.01 (*only department ABC*)*.05(*only salary attribute*)=.0005 of reported changes are of interest*

Assuming each detected change is forwarded through a message to the requesting agent, [Chaw97b][Labi97][Lind86] all send messages where 99.95% are of no value. Finally, although CPU is much faster than I/O, a CPU bound system will be hindered by comparing entire tuples instead of selected attributes. As with messages, approximately

99.95% of the CPU used for comparing attributes is performing useless work. With the SQL-based algorithm, benefits in space, time, and message complexity can be realized.

A further improvement in space and time complexity can be realized by restricting the "copy" to pertinent records and attributes. Given the estimations above, the "copy" of the base relation need only contain the key and the salary fields for a total tuple size of *16* bytes. In addition, the total number of applicable tuples from *P* is, on average, *100*. The total size of the copy is then *1600* bytes, or *1* page. Even without a non-clustered index on *P*, the total I/O cost for each differencing run is at most *N+1* due to the relatively small size of the restricted copy. As the copy also contains the primary key, the total I/O cost for joining back to *P* to detect deletes and updates should be at most *.01(N)+1* if the base relation is indexed on the primary key. If a secondary index is available on the attributes of P that are used to restrict the number of tuples in the snapshot, the cost for insert detection can also be reduced to *.02(N)+1*. Thus I/O is at most *N+1* and may be as low as *.03(N)+2*. Only changes that are of interest are reported, thus, in this example, the total number of messages is *99.95%* fewer than previous techniques.

Query-based differencing can further improve performance by allowing the underlying database system to use its full query processing capabilities. As an example, B-trees with a height of 3 and 3 data pages typically require 5 I/O to read data into memory and thus will cost 10 I/O per relation comparison. For base tables of 3 data pages or less, leaving the underlying tables as heap instead of B-tree can reduce I/O to 6, thus reducing I/O by 40%. Although CPU utilization will be higher in the second case, this tradeoff will typically lead to a 35% improvement in overall system response time. Allowing the DBA to choose storage structures and allowing the optimizer to choose the

best join strategies can improve upon the forced implementations of storage structures required by pre-sorted sort-merge algorithms.

The worst case cost of the MDS algorithm occurs when the data source does not support some type of sorted structure, the replica contains all tuples and attributes, and the relations are too large to fit into memory. In this case, the optimizer will most likely choose to sort the base relation before joining to the older copy. Although the costs to sort then compare are less than generating a report due to the fact that no output file is created, the cost to perform this operation can be as high as 4N. While some portion of the older copy and the base relation may reside in memory, thus slightly reducing costs, the worst case cost is 4N. In such an environment, however, the best case cost of the sliding window protocol is also 4N. If key-valued secondary indexes are available and the memory is of sufficient size to hold *N+1* pages, the total I/O cost can be reduced to 2.2 N if leach indexl is .1N (used to determine *inserts* and *deletes*) and a hash join is used to compare relations for *updates*.

For small tables and highly restrictive snapshots the basic SQL-based algorithm can be used to improve the performance of previous algorithms. But what about tables spanning hundreds of pages or data sources without join capabilities? Data sources without join capabilities will be addressed in section 4.3. For large relations with few restrictions, the bottleneck of locking the base relation while performing differencing must be addressed. Most of the previous algorithms ignore this issue and thus lead to errant performance characteristics for OLTP systems. The SQL-based approach offers several solutions by either eliminating the need to scan the entire base relation or by

incrementally referencing subsets of the data. The level of improvement realized over previous algorithms depends upon the chosen algorithm.

Modified Table-schema Approach

Reducing the cost of scanning the original relation can be accomplished in several ways. If the number of pertinent tuples is small and if a secondary index is available on the attributes that are used to restrict the snapshot, a table level lock can often be avoided by utilizing the index. There are scenarios, however, where *any* change to *any* tuple in the relation must be detected. An alternate approach must be adopted in these cases to prevent locking contention. If the underlying data source and application program cannot be modified, an incremental differencing algorithm can be utilized (see section 4.3). If the data source allows for the modification of table schemas, several approaches are available to greatly reduce I/O and improve concurrency.

R* was one of the first database products to augment schemas with a time-stamp that could be used to enhance change detection [Lind86]. R* used the time-stamp for creating snapshots of pre-defined queries. A snapshot in this sense is the same concept as a materialized view in a data warehouse. With the R* approach, each table that is designated as providing data for a snapshot is internally augmented with a time-stamp attribute and a record identifier. As tuples are modified, the time-stamp is automatically updated. Although this approach reduces the cost of identifying changes needing application to the snapshot, much of the System R* code was rewritten to properly utilize the new fields. To efficiently detect changes, this approach requires knowledge of every available tuple address per page as the snapshot references the record identifier to determine if deletes have occurred. By their admission, this requirement is quite

restrictive. Rewriting the database engine is also not an option for most systems. The R* approach does, however, suggest several techniques that can be adapted to provide good performance for a query-based differencing system.

Simple Time-Stamps (TS)

The basic time-stamp approach has merit if the underlying data source contains table schemas that can be modified to include a time-stamp attribute. The time-stamp field can be either an actual date/time attribute or it can simply contain an integer that is incremented each time a modification is performed. Time-stamps can be used most effectively if the underlying data management system supports auto-incrementing data types. For systems without auto-update capabilities, time-stamps or incremental integers can still be used effectively if all access to the data is restricted to well-defined application interfaces. Controlling database access through the use of pre-defined application interfaces is a common practice in most business environments. In such an environment, application code must be changed so that the time-stamp field is updated at each tuple modification. Inserts can be implemented with null time-stamps or with time-stamps where the appropriate value is pulled from a separate table or a system variable. The non-null time-stamp option leads to a more efficient query by eliminating tri-value logic for nullable key fields.

The problem with this simple approach is its inability to efficiently detect deletions. Although this is a serious limitation, there are many systems that do not allow deletions, but instead require compensating transactions. In this environment, TS is a reasonable approach. For applications that allow deletions, the application code could insert the tuple into a temporary table before deleting from the base relation, but this

unnecessarily lengthens the transaction. Basic time-stamps do not eliminate the need for a copy nor do they prevent the necessity of scanning and joining all rows of the base table and the replicate old copy if deletes are allowed. In this case, the increased size of each tuple augmented with a time-stamp may hinder instead of improve performance.

Time-Stamps and Delete-Flags (TDI)

To fully utilize time-stamps for efficient change detection in systems that allow deletes, the relation must be additionally augmented with a delete-flag attribute. The accessing application must also be modified to change all *delete* statements to *updates* of the delete-flag field. This marking, similar to the dBase [Dunl90] implementation of mark-then-pack for deletes, enables deleted records to be easily detected. The time-stamp with delete-flag algorithm (TDI), instead of the application, physically removes the marked records after checking for changes in a separate step. Modifying the application to use a view instead of the base relation easily restricts application reference to non-deleted rows only. The basic algorithm for this approach is similar to that of R* except that it can be applied to any data source with SQL and time-stamp capabilities. Figure 13 outlines this algorithm and Figure 14 shows the performance characteristics of a general-business-use data source using this algorithm.

We have shown in Figure 12 that a full-outer-join operation must be utilized if inserts, updates, and deletes are allowed. Delete identification capability is not required with this algorithm. Hence, the right-outer-join specification is no longer needed. Given this requirement, the join and left-outer-join have not been modified from the algorithm outlined in Figure 12. In line 9, the base relation is locked by the opening of the cursor until the cursor is closed, thus ensuring no changes occur during the looping of the

algorithm. If changes are found, the SQLCA.rowcount will be greater than 0. If no changes are discovered, the algorithm terminates immediately. After each row is fetched, the type of change is detected and the appropriate modifications to the data sources are affected. After each fetch, a check is used to determine if any more rows exist. When no more rows are available, the algorithm terminates and the cursor is closed to release the locks on both the base and duplicate relations. All changes are applied as part of a single transaction as committing between updates closes the cursor. The benefit of this approach is that errors are automatically handled by the underlying DBMS system.

```
[1] Create cursor CHANGES as
[2] Select t1.f1,t1.f2,…,t1.fn,t1.delete_flag,change_type='Update' from basetable
    t1,duplicate t2
[3] Where t1.time-stamp>last_time_checked
[4] And t1.key=t2.key And ((t1.f1!=t2.f1 or t1.f2!=t2.f2 or …t1.fn!=t2.fn) or
    t1.delete_flag='T');
[5] Union
[6] Select t1.f1,t1.f2,…t1.fn,t1.delete_flag,change_type='Insert' from basetable t1
[7] Where t1.time-stamp>last_time_checked
[8] And not exists (select t2.key from duplicate t2 where t2.key=t1.key);
[9] Open cursor CHANGES
[10]    Fetch CHANGES into structure;
[11]    While SQLCA.rowcount>0  { //stop when no more rows available
[12]    If structure.delete_flag!='T' { // Send structure to ATP before each
    "duplicate" modification
[13]      If change_flag='Update' { // Change was an update
[14]    Update duplicate set f1=structure.f1,…,fn=structure.fn where
    duplicate.key=structure.key }
[15]      Else {  //Must have been an insert
[16]        Insert into duplicate values structure  } }
[17]    Else {  //structure is a deleted tuple
[18]    Delete from duplicate where structure.key=duplicate.key
[19]    Delete from basetable where current of cursor  }
[20]    Fetch CHANGES into structure;}
[21]    Close cursor CHANGES;
```

**Figure 13: Time-stamp with Delete-Flag Algorithm**

IO Cost to Identify Changes (scan of $B_a$) = $S$
Cost to join $B, C$ to gather old/new pairs - $\Delta$ (N)
TOTAL IO Cost TS = S+$\Delta$(N) //assume some type of key on $C$
PROBLEM: Locking all of B while scan is occurring can degrade concurrency

**Figure 14: Performance Analysis of TS&DF Algorithm**

The basic TDI algorithm is easy to implement. At defined intervals, the SQL statement in Figure 13 is issued against the data source. The SQL groups changes into two categories, inserts and updates. Two groups are used instead of three to improve the efficiency of the query. An insert is detected if the time-stamp field is greater than the last time the algorithm was run and the same key does not exist in the "replica." An Update is detected if the time-stamp is greater than the last time the algorithm was run and the delete-flag attribute is set to 'T' or the pertinent fields have changed. Recall that the delete flag is set to "T" in the application program in leau of the application code issuing an SQL delete. If the delete flag is set to "T", the update is really a delete and is treated as such. If the "delete flag" is not "T", the change is an update and the values are propagated to the snapshot. The large improvement in performance over previous algorithms is achieved by eliminating the need to compare rows to identify changes.

Assuming the non-existence of a time-stamp index on $B_a$, a scan of the base relation is required. Presuming the existence of some type of key on C, the cost of identifying matching records in C is $\Delta$. While the Yao [Yao77] function would argue that changing 1% of the tuples requires accessing more than 1% of the pages of a table, actual implementation suggests just the opposite to be true. If the data source supports large tables, it is likely that the data source is some form of database product. Products such as Excel or Lotus 1-2-3 are queriable, but the data set size is comparatively small.

Even with these products, if the data set spans multiple pages, most changes are grouped due to users adding or removing "sets" of like records at any given interval.

For database products, very few data structures are supported. In the relational arena, the structure of choice is B-tree, with a few vendors also supporting heap, ISAM, and Hash structures [Ingr91][Info97a][Micr97][Orac94][Syba96]. With heap structures, all inserts appear at the end of the data set. The records that are most likely to be changed are those that have been recently entered incorrectly or are those that are purged to reduce the size of the table. In either case, the changes are isolated to the first or last few pages of the relation. ISAM and B-tree structures also place all inserts at the end of the structure if an incremental key is being utilized. While it is possible to have deletes and updates issued against older tuples, most changes to tuples more than one month old are considered changes to historical data. Seldom does historical data need to be changed for business applications. Even if an incremental key is not being utilized, some other attribute can generally be utilized to group changes into categories. As the replicate table can be indexed on any attribute(s) that seem reasonable, the grouping attributes can be used to maintain likely modified tuples on the same page. Keying the replicate table on probable grouping attributes eliminates any distribution problems caused by using a Hash structure on the base relation. For systems, such as personnel applications, where modifications are more random in nature, the number of changes between queries seldom forces access to a large number of pages. Thus, while it is possible for a 1% change to require access to greater than 1% of the pages, it is equally likely that such a change will access less than 1% of the pages.

To further improve performance of the basic algorithm, the cost of scanning the base relation can be greatly reduced if the base relation is enhanced with a secondary index on the time-stamp attribute (TDI). As an example, Figure 15 illustrates the performance improvement gained by introducing the secondary index. Finally an additional improvement is outlined in Figure 16 where a secondary index is added that contains the time-stamp and the pertinent fields of the snapshot (ATDI).

Placing additional fields in the secondary index increases I/O for updates to pertinent attributes. As "inserts" and "deletes" already affect the secondary, the cost is not appreciably increased for these operations. If all fields in the primary are included in the snapshot, this algorithm performs worse than the simple secondary index approach. For highly restrictive snapshots, however, differencing costs can be greatly reduced. For instance, if the snapshot contains only 30% of the attributes, and a change to 1% of the tuples occurs between each differencing run, I/O costs can be reduced to $.014N$ by eliminating the need to join back to the base relation as shown in Figure 16.

Astute readers will note that the highest cost of the augmented index algorithm is the cost of I/O on the old relation. One way to reduce I/O cost is to restrict the old copy/snapshot to the applicable fields and rows requiring change detection as assumed by the calculation in Figure 16. If both the snapshot and the index are projections of the base relation, the cost of comparison can be greatly reduced if the sum of the size of applicable fields is relatively small. As an example, many relations contain a large text field to hold extraneous data. Eliminating this "comment" field from consideration for update detection when the size of the comment field is greater than the combined size of all other attributes in the relation can reduce the cost of change detection by more than 50%.

Figure 17 outlines the cost of utilizing such a technique with the ATDI algorithm for a typical business application.

IO Cost to Identify Changes in Index (Secondary Btree Index on TS) = I*Δ
Cost to join index, $B_a$ = S*Δ // Need to join index back to augmented base relation
Cost to join *C,B* to gather old/new pairs ≈ Δ(N) // Need to join identified changes
$Cost_{TDI}$ = S*Δ + Δ(N) + I*Δ+ K // with old Copy
PROBLEM: If a large percentage of the table changes, the cost of using the index can exceed the cost of a base table scan. Note: There is a low likelihood of index-level lock if only 1% changes. It is, however, possible to acquire a table level lock due to the random distribution of changed tuples (Yao function).
Increase in space complexity = *(S+I)-N+N (size of C) = S+I*
For relations where T≤A (highly unlikely,), (S+I)≥3N. Worst case costs with a B-tree is of height K are:
*Max(TDI) = ((I*Δ)+K)+((S*Δ)+K)+((N*Δ)+K)=((I+S+N)*Δ)+3K*

**Figure 15: ATDI Algorithm Costs using a secondary index**

I=1+X/(P/(A+size(tid)+K+F)) // Index includes TS, DF, tid, Key, and pertinent fields
Increase in space complexity = (S+I)-N+N = S+I
Cost to identify change in the index = I*Δ + K
Cost to join back to base = 0 – not needed for algorithm
Cost to match with old copy to gather old/new pairs - Δ(N)
$Cost_{ATDI}$ = *I*Δ + Δ(N) + K*

**Figure 16: ATDI Algorithm Costs using an augmented secondary index**

*I=1+X/(P/(A+size(tid)+K+F))*
Increase in space complexity = *(S+I)-N+W = A+I+W*
Cost to identify change in the index = I * Δ
Cost to join back to base = 0 – not needed for algorithm
Cost to join *I,Cr* to gather old/new pairs - *Δ(W)*
$Cost_{ATDI_PC}$ = *I*Δ + Δ(W) + K*

**Figure 17: ATDI Algorithm costs in conjunction with a Projected Copy**

Tuple Level Checksums (TLCS)

Although time-stamps and delete-flags can be utilized to achieve adequate performance, not all applications can be modified to utilize the new attributes. One

technique that can be used to improve performance of a system with an unchangeable front-end, like a legacy Cobol application, is that of augmenting each tuple with a checksum field whose value is based upon pertinent attributes – those attributes having relevance to the snapshot. Use of a checksum value can also be implemented by creating-then-joining to a separate table that contains only the key and the checksum attribute. Modifying the base relation will slightly improve performance by eliminating the need for an additional join.

After the relation's schema is modified, a separate job must be run against a quiet system to calculate and update the value of the checksum for each tuple. As the system becomes active, a differencing procedure is run at specified timer or resource driven intervals to calculate the checksum and compare it to the existing checksum attribute of the tuple. If a discrepancy is detected, the base table checksum is modified and the record is joined to the copy to yield an old/new value pair. The total I/O cost per iteration is

$N+\Delta(W)$ *where* $W < N$

If the number of attributes in C is one-tenth the number in B (a few pertinent attributes), the I/O cost can be quite low. (E.g. it is $1.001(N)$ if a 1% change is assumed and C is appropriately indexed.) If N is less than 100, tuple level checksums perform reasonably well. For tables spanning more than 1000 pages, however, locking the entirety of the base relation prevents the algorithm from being executed on a frequent basis. Although this technique improves upon the previous algorithms where application modification is prohibited, it does not yield sufficient update gathering performance for an asynchronous trigger processor or data warehouse history gathering system due to high

locking overhead when large relations are involved. Frequent queries against large tables may maintain an almost constant lock on the relation.

A variant of this approach can be used if the local site does not support joins or if it does not have sufficient disk space for the replicate copy. In such an environment, the change detected by the checksum is queued and the data is forwarded at intervals for joining at the remote site. The queuing reduces message traffic and can improve join performance. Remote site differencing is further explored in section 4.6.

Incremental Differencing

The Modified Database Schema (MDS) and the Tuple-Level Checksum (TLCS) algorithms both require scans of the base relation to identify changes. This scanning must be performed unless triggers, replication, an intermediate layer modifier, an index, or an application change is introduced. For large tables, scanning the entire relation during peak hours is not a reasonable option due to locking contention. The Incremental Differencing Algorithm (IDA) introduced here presents a plausible solution to this problem by utilizing a looping construct to compare incremental subsets that comprise the entirety of the base relation.

The basic incremental differencing algorithm is found in Figure 18. The IDA algorithm makes use of an auxiliary table to help in partitioning the data into appropriate subsets. Before the IDA can be started, an initialization step must be run to calculate the number of pages touched per range of clustered key values. This information is stored in the auxiliary relation. The presumption of some form of clustered or ordered key is quite reasonable given that a table spanning thousands of pages does not yield sufficient performance unless some form of key is available. If the base relation is heap with a

secondary (non-clustered) index, the number of pages referenced by the p-tids (tid pointers) of the index must be calculated. If the primary or secondary is of hash structure, with no clustering or ordering index type available, the range query parameters generated by the initialization step will be of little use, as a table level (or segment level) scan will be performed for range queries. For this discussion, we assume the existence of a B-tree or ISAM-like indexing structure.

```
Select max_counter=max(counter) from auxiliary a
Where a.table_name = 'my_table';
I=0;
While I<max_counter {
  I = I+1;
  Query data from my_table in the range specified by the start and end
  Values found in row I of the auxiliary relation. Compare tuples from
  this range to the same range in the older copy of the relation.
  Apply changes to the old relation.
  I=mod(I,max_count);
  Wait(x seconds);
}
```

**Figure 18: Basic Incremental Differencing Algorithm**

The comparison steps of this algorithm are the same as the MDA algorithm and are not reiterated here. As the "I" values within the loop range from 1 to max_counter, we can presume the correctness of the algorithm if the initialization of the setup table is performed correctly. The initialization step of the IDA must scan the entire relation looking for values that partition the data into ranges that span a pre-specified number of pages. Partitioning the data by the average cardinality is not sufficient due to the possibility of unequal key distribution. The number of pages in each range (histogram statistics) must be stored to ensure that table-level lock escalation does not occur. As each range is identified, an entry is placed in an auxiliary table with the boundary values of the range as well as an incremental counter to uniquely label each range. After each

range tuple is inserted, the page counter is reset, the start-range key value is set to the end-range key value of the previous insert, and the scan of the base relation continues. At the end of the scan, the final range is left with an open-end range to ensure detection of incrementally increasing keys. To simplify the algorithm, the initial start range and the final end range are inserted with the absolute minimum or maximum value available for the data type of the key. This eliminates the need to modify the query within the algorithm for the boundary instances. The auxiliary table is then used in the basic algorithm to formulate successive queries against the base relation. This partitioning of the differencing transaction into subset comparisons is similar to the Mini-Batch technique outlined by Jim Gray [Gray93]. As the distribution of values changes within the relation, the initialization step can be re-run to modify the number of iterations as well as the range values of successive runs. The *wait* function can be modified to be purely timer driven or it can be tuned to incorporate load-balancing information.

Local Site Incremental Differencing (ID)

While Figure 18 outlines the basic algorithm to incrementally gather subsets of rows of the base relation, it does not detail the comparison to the "old" copy of the relation. The old copy is necessary to determine both the existence and the nature of the change that may have occurred in the specific range. If the local site is such that it allows for the creation of additional relations, both the auxiliary table and the "replicate" copy can be maintained within the same "database" as the base relation. If all tables are maintained locally, a three-way join can be used to determine differences using a modified version of the algorithms in Figure 12 and Figure 18. As each union in the basic Query-Based Differencing Algorithm contains the range specification and a

modified "where" clause that references only pertinent attributes, the algorithm does not suffer from locking problems nor does it report spurious modifications. Locking does not escalate due to the key restrictions in the range table. In the worst-case scenario where all attributes must be maintained in the copy, the cost of each complete table comparison is *2N+size (auxiliary table)*. As an example, with a 1 million row table having a tuple size of *200*, the size in pages of the auxiliary table is *(N/pages per range)/(P/(2*(size (key))+4))*. Assuming a key size of *8* bytes, a range of *10* pages, and a page size of *2048* bytes (i.e. INGRES), the size of the auxiliary is

*(100,000/10)/(2000/20)=10000/100=100 pages or approximately .001N*

The worst-case cost assuming some type of index on the base relation is *2.001(N)*. This can be reduced at the expense of CPU if the copy is compressed. It is also highly unlikely that the copy need contain all attributes of the original. If using this technique with an ATP and the triggers are simply "on insert" or "on delete", only the key values need be maintained in the copy, thus reducing I/O by *.9N*. Seldom does the list of reasonable triggers within the ATP reference all attributes (necessitating the maintenance of all attributes in the copy). Knowledge of defined triggers can be used to modify the size of the *duplicate* copy.

The overwhelming benefit of incremental differencing is that it can be run even during peak hours to identify changes within a reasonable time window. One significant pitfall is that updates to key values will be incorrectly identified as a delete followed by an insert. This is true, however, for all of the Non-SQL as well as the SQL approaches outlined except the augmented TS/DF algorithm.

The basic incremental algorithm solves many problems and makes differencing on large legacy systems feasible. It does not address situations in which the underlying database schema cannot be modified nor is it appropriate for databases that do not support joins. Some legacy systems do not allow for the introduction of multiple tables into a database. On other systems, even if the modification is technically allowed, the data source does not have sufficient disk space to support both the original and a copy of the base relation. For scenarios such as these, an incremental remote-site differencing scheme is proposed.

Remote-Site Incremental Differencing with Checksums (ID/RCS)

If the local site does not support more than one table, or if it does not have sufficient space for the duplicate relation, a remote site can be used to store the extra table(s). The remote data site should typically be supported by a join enabled database with good query processing features. At initialization, a partial or full copy of the base relation is shipped to the remote site. The remote copy is then indexed on the keys used in the range queries to efficiently process comparisons when partial snapshots are passed from the base relation. A simple illustration of this approach is found in Figure 19.

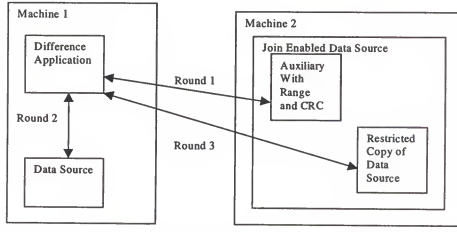


**Figure 19: Illustration of Remote-Site Incremental Differencing (RID)**

While the basic incremental differencing approach of polling subsets based upon values in the auxiliary table is feasible in such an environment, the constant flow of data between sites can degrade system performance by overflowing the network bandwidth. Although the cost of scanning the base relation cannot be reduced (assuming the data source is not modifiable), the number of messages required between the remote and the local site can be lessened if the likelihood of a change within a single range of data is relatively low. Reduced messaging costs between the base and the remote sites can be realized by calculating a modified checksum on the pertinent returned fields within the range query and comparing that value with a pre-calculated range-value checksum stored within the auxiliary relation. Jim Gray [Gray93] has suggested the use of block level checksums for data validation in other arenas and Comer [Come91] has outlined a basic checksum algorithm that could be used for such a system. The use of several checksum methods can reduce the likelihood of a missed update.

The RID algorithm works as follows. When the range specification is selected from the auxiliary relation to build the query against the base table, the former checksum for that range is also retrieved (round 1). At the source site, the checksum is calculated for data within the range as the data is retrieved (round 2). If the values are identical, no message is returned to the remote site. If a change is detected, the modified checksum value and the tuples within the range are forwarded to the join-enabled remote site to identify the exact nature of the change (round 3). While the Yao [Yao77] function indicates that referencing a randomly selected, small number of tuples quickly escalates to accessing all pages of a table, it does not suggest such an escalation if the selected tuples

are identified in a more structured manner. As most applications use some form of clustering key, data is typically structured such that 1 - $\alpha$% of the changes are propagated against the most recent $\alpha$% of inserts for some $\alpha$ (Zipf's Law). In this case $\alpha$ will typically be 20%. In such a scenario, the use of range checksums will effectively eliminate the need to pass 80% of the relation to the remote site during each incremental relation scan. Moving the tuple comparison and the copy to a remote site also reduces CPU and I/O contention at the source site. The use of multiple sites is especially applicable where data warehousing is implemented.

View Based Differencing (VBD)

While the query-based algorithms are specifically designed for use with an ATP, their applicability to many other areas should be quite evident. A recent hot topic has been that of data warehousing and view maintenance in data warehouses. Much work has been dedicated to identifying efficient techniques for view maintenance ([Labi97] [Quas96] [Zhug96] [Zhug96a]). In a data warehouse, a materialized view is similar to a snapshot in that it represents a materialized result of a query performed against relations in the source database(s). Refreshing the data in a given warehouse view (snapshot) has been accomplished by using one of two basic approaches. If an application or report is written to calculate a "snapshot," the data is selected, projected, and possibly joined together then output into a file. After file creation, either the entire file is uploaded into the data warehouse or a "diff" is performed against the two versions of the output file to identify changes. Any changes identified are then applied to the warehouse view (snapshot). Without an application, all data from all involved relations must be copied

into flat files. After loading the flat files into the data warehouse, all pertinent snapshots are re-materialized within the remote database. The new view can either replace the older version or it can be compared to the old snapshot to identify changes that must be applied to the old copy. In either case, the cost of managing a single view can be staggering if there are large relations with only a few interesting views.

Query-based differencing can be used to greatly reduce the cost of determining changes by "materializing" the view (query) in a SELECT operation at the base site before forwarding information to the warehouse. Source-site view materialization is especially critical where relationship tables are necessary to complete a many-to-many join between two relations. As the view is processed at the source site, possibly several relations can be eliminated from the copy-then-compare algorithms used in instances where view materialization occurs within the warehouse. Even without time-stamps, the use of range or tuple-level checksums can greatly reduce the I/O required to identify a change that affects a specific snapshot. Because the changes are applied to the warehouse snapshot instead of replacing the entire warehouse view (relation), I/O and downtime within the warehouse are greatly reduced. If the specific warehouse view is concerned with a limited subset of the relations, the reduction in I/O can be quite dramatic. Assuming a 1% change between each differencing scan, the packing of changes into a page before passing to the remote site can reduce messaging costs by greater than 99%.

As an example, consider 4 relations involved in a given view, each containing 100,000 tuples with an average of 10 tuples per page ($N = 10,000$). Without the aide of an application, the cost of identifying and applying the changes to the warehouse, assuming a 1% change where 50% of the attributes are in the final view, will be quite

high. The costs will be either *16N+14N+2(.01(2N)) = 300,400 I/O* to apply only changes to the view or *16N+4N+2N= 220,000,* assuming sort-merge joins are used, to replace the view. These costs are calculated as follows: 4N to read from the source database, 4N to output to a flat file, 4N to read from the flat file, 4N to copy into the remote database tables (the warehouse), 4N to read and join within the database (assuming B-tree structure and ignoring index overhead), 2N to write results to a flat file, 2N to read in the old file, 2N to read in the new file, .01N to write changes to a separate file, .01N to read changes into the database, 2N to read in the old copy from the database, and 2N to write the updated copy back into the database). For view replacement, the second round of writing to a flat file for differencing is eliminated, but the data within the data warehouse is unavailable for a longer period of time.

Several algorithms can be used to facilitate view-based differencing. If local-site view materialization uses a tuple level checksum to identify changes and a keyed structure is available for all tables, the cost of VBD is considerably lower than previous algorithms. Given an efficient structure on the 4 relations being joined, the cost to create a change file is 4.06N (4N to read in the relations plus 4ΔN to join to other relations plus an additional 2ΔN to output the change file). While this cost could be greater given poor storage structure, it could also be less if good secondary indexes are available or some portion of the relation(s) is already cached within the server. The generated change file, of size .02(N) is sent through RPC calls to the warehouse. Applying the changes to the warehouse will require at most 4N (2N for both reading and writing within the warehouse), and it is possible for the costs to be as low as Δ(2N) = .02(N) if the changes

are clustered around specific keys. Thus, in the best case, the cost is reduced to $4.1(N) = 41{,}000$, reducing the I/O cost by over *70%*.

If the relations contain time-stamps and delete-flags within a secondary index, a variation of the TDI algorithm (see Figure 2) can be used to efficiently generate a change file that can be applied to the warehouse view. As the changed tuples within each relation must be joined to the other 3 relations, the total cost of identifying the changes in this example is 4*(3*(.001*1.1)) = .132(N) assuming reasonable clustering of data. As these changes do not *require* that they be written to a flat file, an RPC call to the warehouse can transmit the changes within a few packets. The cost of applying the changes is the same as above with a worst-case cost of 4N and a best-case cost of .02(N). Thus, the "best-case" performance of VBD without restricting the input specifications of the join is .152(N), or 1520 I/O for this example. If the snapshot is a highly restrictive one, the performance improvement is even greater. When dealing with gigabytes of data, this improvement can reduce the time required to reload the warehouse from hours to only a few minutes.

# CHAPTER 5
# PERFORMANCE EVALUATION

The query-based differencing algorithms provide good theoretical results. Without implementing some of the algorithms and running them against commercially available products, a consistency between the theoretical and actual results could not be ensured. Hence, a representative algorithm was selected from each broad category of differencing techniques and approximately 120 tests were conducted for each selected algorithm. Additional queries were performed to guarantee warm start results. During implementation, a few anomalies were discovered, but they were such that it demonstrated that the new algorithms reduce I/O even more than previously outlined due to some caching issues.

For performance testing purposes, the algorithms were implemented on a Dell Dimension XPS with a Pentium Pro 200 processor and 64 Meg of memory. The machine utilized the Windows NT operating system and the chosen database product was INGRES II from Computer Associates. For the INGRES II testing, the native ISQL interface was used to run the queries and return the performance results. The schema of the base relation used in these tests was a modified version of the Customer table layout as specified in the TPCD [TPCD98] benchmark. Running the algorithms against a Microsoft Excel spreadsheet using an ODBC driver also tested the applicability of these algorithms to other environments. While correct results were returned from the Microsoft Excel tests, no performance measurements were recorded.

The number of parameters that could be varied to ensure a valid comparison between previous algorithms and the new algorithms were quite numerous. To reduce the complexity of the results, only four algorithms were compared: ATDI - Augmented Timestamp with Delete flag, Indexed; MDS - Modified Data Source; P/SWC - Presorted or Sliding Window Comparison; and RSBC - Report and Sort Before Comparison. The other parameters that were chosen to be varied include the number of tuples in the base relation, the number of attributes and tuples in the replica, the percentage of tuples changed between each differencing run, and the distribution of the modifications throughout the relation. Only the I/O costs are evaluated, as the I/O cost component is the dominant factor in wall-clock speed.

Varying Size of Replica

One of the drawbacks of previous approaches to differencing was the necessity of either comparing two complete sets of data or running a report to reduce the size of the original before comparing [Labi96]. In either case, the amount of I/O is prohibitive if only a subset of rows or attributes or a restricted combination of rows and attributes is of interest. To demonstrate this fact, and to show implementation results of the new algorithms, a suite of tests were performed in which one of several variables were altered. The results were obtained using base table sizes of 1000, 10,000, 100,000, and 500,000 rows. With the exception of the scale on the Y-axis, the graphs are quite similar for all plots. The percentage of change between iterations was held at 1%. The graphs of the 100,000 tuple results are found in figures 20, 21, and 22.

As can be seen in these Figures, the ATDI and MDS algorithms performed significantly better than P/SWC and RSBC. Of particular interest is the fact that the

inherent caching nature of the database server enables the ATDI algorithm to achieve an I/O cost of less than 2 regardless of the size of the base relation or replicate copy [Alon90].

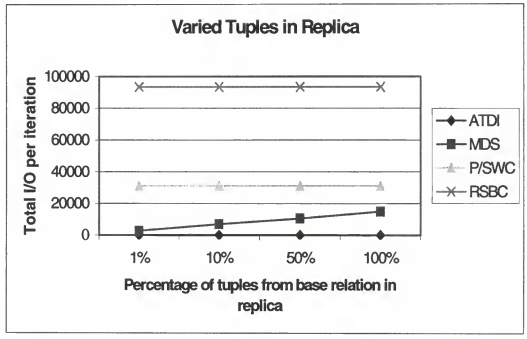


**Figure 20:Varying % of Base Relation Tuples found in Replica**

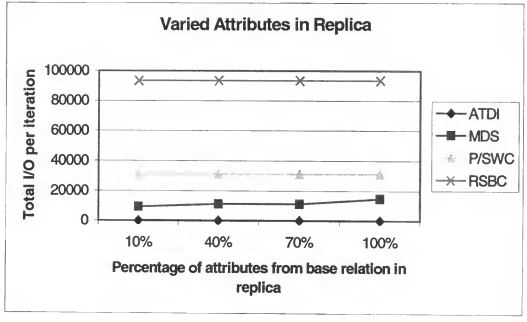


**Figure 21: Varying % of Base Relation Attributes found in Replica**

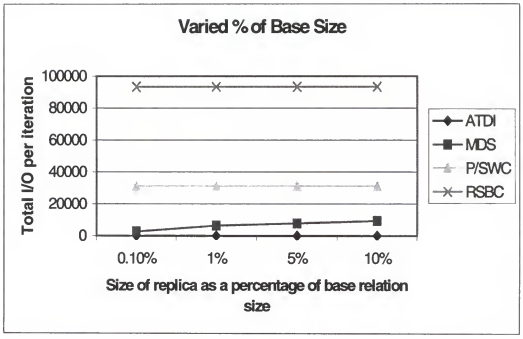


**Figure 22: Varying Replica Size as a % of Base Relation**

Figure 20 shows the performance of the algorithms where the number of tuples in the replica is restricted to a percentage of the tuples in the base relation. This type of scenario is common, for example, when the triggers or warehouse may be interested only in persons in a single department or in people with a salary greater than a given threshold. As the number of tuples in the replica increases, there is a linear increase in the number of I/O required to identify the change set. The 2N I/Os expected for MDS was reduced to .9N due to data caching within the server as well as efficient use of available secondary indexes.

As demonstrated by the non-linear curve of the MDS algorithm in Figure 21, varying the percentage of attributes leads to a stair-step curve due to the existence of non-usable space on each data page. Eliminating some attributes from the comparison data set may have little effect for it does not yield sufficient space savings to incorporate

additional tuples onto a data page. Selectively removing other attributes from the change set may lead to a drastic reduction in the size of the duplicate relation's footprint by allowing for additional rows to fit on a single page. As an example, assume a data storage product that uses 2000 byte pages. Given a relation with 10 attributes, with the $10^{th}$ attribute declared as "char(500)" and a total tuple size of 1400 bytes, the total number of tuples that can reside on a single data page is 1. Without the use of compression, the total number of pages in the relation will exceed the number of tuples in the relation due to the index overhead associated with most storage structures. In addition, 600 bytes of each page will be wasted space. Removing the $10^{th}$ attribute yields a tuple size of 900 bytes, thus providing enough space in the duplicate relation for two tuples on each page and effectively reducing the size of the footprint of the replica by 50%. It is important to note, however, that after the reduction, 200 bytes on each page are still wasted – *(2000 – (900*2))*. For the sake of argument, assume the remaining 9 attributes are equal in size and thus are 100 bytes each. Removing another attributed from the duplicate relation yields a tuple size of 800. Removing this attribute increases the amount of wasted space on each page, but does not yield sufficient space to allow for additional tuples to fit on a given data page. The same is true even when a third attribute is removed, as 3 tuples of size 700 bytes requires 2100 bytes and the data pages are restricted to 2000. It is only after removing a fourth attribute that a further reduction is achieved. Using products that allow rows to span pages or using compression to remove white space from the tuple may not yield the same results. Figure 22, which graphs the I/O required when the replica is restricted both vertically and horizontally, shows essentially the same curve as basic tuple restriction.

Varying the Number of Changes Occurring Between Iterations

To ensure that the new algorithms perform at least as well as the previous algorithms in all cases, several tests were performed in which the size of the replica was equivalent to the size of the base relation. During these tests, the number of tuples modified was varied from 1% to 100% of the base relation. As the new algorithms are designed to run on a frequent basis, the possibility of greater than 2% of the base relation being modified between iterations is extremely small. Figure 23 outlines the results of these tests.

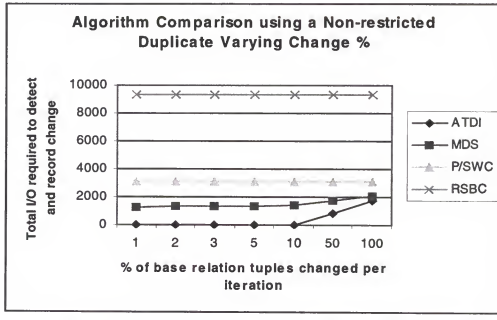


**Figure 23: Algorithm Comparison Using Full Duplicate varying the number of changes issued between iterations.**

The ATDI and MDS algorithms still significantly out-perform the previous algorithms even in this worst-case scenario. The changes distribution was comprised of 80% inserts and 20% updates where the updates were performed against the most recent set of inserts as determined by the incremental integer key. When 100% of the tuples are

changed, ATDI and MDS yield the same results as the time-stamp attribute no longer restricts which rows must be checked for determining the existence of a change. One item worth noting is the sudden increase in I/O cost for the ATDI algorithm when the change percentage increases beyond 10%. The secondary index on the time-stamp attribute becomes less useful when larger percentages of the base relation must be joined with the secondary to gather all attributes required for comparison to the duplicate relation. Due to the benefits of caching within the server, the worst-case scenario for the new algorithms, even when 100% of the tuples have changed, yields 1.2N I/O as opposed to the 2N required by P/SWC where N is the size in pages of the relation. As the size of the relations increases, the percentage difference between the two algorithms will decrease.

ATDI Varying Change % and Distribution of Modifications

The ATDI algorithm, unlike the other algorithms discussed here, does not use pure comparison to identify both the existence and the type of change. Recall that the identification of update descriptors is a two-stage process – identify that a change has occurred, then identify the nature of the change. Once the change has been identified, the update descriptor must be passed to the appropriate update descriptor processor. The need to both identify and pass the full update descriptor leads to an opportunity to fine-tune the ATDI algorithm in such a way as to minimize the total I/O required for the full update descriptor delivery protocol.

The variations of the ATDI algorithm have been designated as ATDI1 and ATDI2. The only difference between the two algorithms lies in the way Update-generated change descriptors are synthesized. The ATDI2 algorithm uses a unioned

query with joins between the base and the duplicate relation to identify in one pass the old/new value pair of the update descriptor. The identified update descriptors, which contain old/new value pairs, are inserted into a temporary table for persistence. The ATDI1 algorithm uses a two-pass approach to calculate the update descriptors in which the first pass identifies the changed tuples by using the timestamp index. A copy of the identified rows is inserted into a temporary relation. The second phase of the ATDI1 algorithm is then used to concatenate the old value onto the tuples in the temporary relation that are identified as being Update-generated update descriptors. For the general case, ATDI1 appears to be slightly better because it eliminates the need to join all of the update descriptors to the replica since Insert and Delete generated update descriptors do not need to be joined back to the duplicate relation. Figure 24 provides a comparison to highlight the difference between the two algorithms. It is important to note that the ATDI algorithms have been condensed from the algorithm shown in Figure 13 to improve efficiency and simplify the output. The algorithms may be slightly more difficult to follow but they do perform the same functionality as the TDI algorithm in Figure 13.

There are other subtle changes in the ATDI algorithms that may not be immediately apparent. The first enhancement to the ATDI algorithm comes with the introduction of the *Distant_future_date* variable. With the ATDI algorithms, there is an assumed secondary index on the timestamp attribute. To avoid the use of tri-valued logic, and to eliminate the introduction of NULLS into a keyed attribute, a distant future date was chosen to indicate the insertion of new tuples. If NULLS are used, the speed of the index scan is dramatically hindered. If a future date is used, all changed tuples are clustered at the end of the index as the timestamp value never decreases. This logic

effectively eliminates the need to scan the index, and instead allows for the use of just the last few pages. The drawback of this approach is that readily available system timestamps cannot be used to identify changed records but instead requires the application to populate the timestamp field appropriately. For new applications this should not be problematic, however the modification of older applications is often discouraged.

ATDI1 Algorithm

```
/* Identify all changed tuples */

Insert into update_descriptor_table
(new_f1, new_f2, …, new_fn)
select f1,f2,…,fn from basetable
where timestamp>Last_time_checked;

/* Flag Insert and Delete tuples with
   the appropriate change type */

update update_descriptor_table set
change_type=
get_D_or_I('DI')
where d_flag='T' or
timestamp=Distant_future_date;

/* Remaining tuples are updates. Must
concatenate old_value onto tuple */

update update_descriptor_table u from
duplicate d set old_f1 = d.f1,
old_f2=d.f2,…, old_fn=d.fn
where u.key=d.key
   and u.change_type not in ('I','D');
```

ATDI2 Algorithm

```
/* Identify all changes along with the
old value pair in one unioned query.
Eliminates the need for the update steps
in ATDI1. Here we assume
applicability of all attributes to simplify
the query. */

/* Identify Update and Delete changes
   in the first query of the union and
   gather Inserts in the second */

Insert into update_descriptor_table
(new_f1,new_f2, …, new_fn,
old_f1,olf_f2,…, old_fn,change_type)
Select b.f1,b.f2,…b.fn,d.f1,d.f2,…d.fn,
get_D_or_U('DU')
from basetable b, duplicate d
where b.timestamp>Last_time_checked
and b.timestamp!=Distant_future_date
and b.key=d.key
Union
Select b.f1,b.f2,…b.fn, Null,Null,…,
Null, 'I')
Where timestamp=Distant_future_date;
```

**Figure 24: Comparison of ATDI1 and ATDI2 Algorithms**

Without the use of the *Distant_future_date* variable, ATDI2 requires one probe of the duplicate relation for every changed tuple in the base relation to determine if the change is an insert or an update. In addition, if the duplicate relation resides at a different locale as the base relation, each tuple changed requires a send and receive message to and from the location of the duplicate relation. Even without the use of the *Distant_future_date* variable, ATDI1 isolates all changes in one step. Delete records are already identified with a delete flag of 'T'. After the issuance of the update to gather the old-value(s) from the duplicate relation, any remaining tuples without and old-key and without a delete flag of 'T' are identified as inserts. The additional benefit of this approach is that the update_descriptor_table can be shipped to a remote site for performance of the "match old value(s)" step without hindering functionality of the rest of the algorithm.

In addition to these subtle changes, several non-ansi SQL clauses were used both for simplification and for performance reasons. The get_D_or_I and the get_D_or_U functions are presented in an abstract instead of supplying the substring functions that are used by each database product to accomplish the desired task. As an example, selecting 'D' or 'I' in the INGRES product suite can be accomplished by the *shift('DI',1-locate(d_flag,'T'))* query. In addition to the get_D_or_I functionality, ATDI1 makes use of another non-ansi compliant feature through the use of *update x from y*. While this query is not standard SQL, the functionality to update one relation from another is available in a large number of commercial products.

The graph in Figure 25 also brings out one additional point. For this test, the type and distribution of change was varied from localized to a forced distribution across the

data set. While we argue that in the vast majority of cases, the changed records will be concentrated in one location of the table, we forced the distribution of updates to determine the affect it would have on performance. A "mod" function was used to force the change of every $N^{th}$ tuple. The number and distribution of the affected pages can be easily calculated by dividing the number of tuples per page by the appropriate change percentage. The distributed update lines of Figure 25 clearly show that a purely distributed update pattern, as may be found with a random key generator, increases the IO cost of our algorithm dramatically. When a mix of 80% inserts and 20% updates is issued instead, especially when the updates are targeted to the more recent data as was done in our tests, the performance of the ATDI algorithms in dramatically improved. The reasoning for using this type of update pattern is that, for most types of business applications, the most recent inserts (e.g. orders) are the tuples that are most likely to require a change. An insert-only change pattern, not pictured in Figure 20, yields the same results as the 80/20 mix.

Figure 25 also brings to light several subtle nuances of the ATDI1 and ATDI2 algorithms. As a larger number of tuples are changed, there exists a crossover point at approximately 2% change in which the ATDI1 algorithm starts to outperform the ATDI2 algorithm due a limited size cache within the server. Recall that the basic ATDI algorithm uses a timestamp index to identify changed tuples. When the changes have been forcibly distributed throughout the relation, an inordinately large number of pages are affected. Recall that the number of pages touched by the distributed update is (page size)/(tuple size)*(change percent). Thus, for a 2% change in our customer relation, the number of pages affected is *int(2000/240)*.02=16%* of the pages in the base relation, or

approximately every 6th page. As the replica and the base relation are essentially structured in like manner, 16% of the pages in the replica must also be accessed.

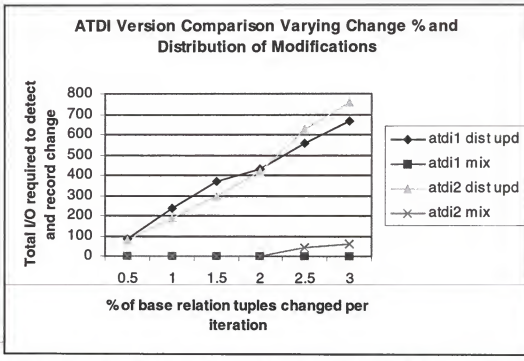


**Figure 25: ATDI Version Comparison Varying Type and Distribution of Updates**

As the number of changes to the base table increases, so does the number of probes into the duplicate relation that are necessary to isolate the type and nature of the change. As the size of the update descriptor table in ATDI1 is quite small in comparison to the number of pages in the duplicate relation, the cost of the ATDI2 algorithm escalates more quickly as the number of pages accessed exceeds the size of the cache. In the lower end of the graph, when the number of pages accessed fits into allocated data buffer memory, the ATDI2 algorithm performs slightly better than ATDI1 as ATDI2 does not require the reexamination of the update descriptor table. When the number of pages accessed by ATDI2 minus the number of pages that can be retained in memory exceeds

the size of the update descriptor table, the performance of ATDI1 surpassed that of ATDI2.

MDS Varying Change % and Size of Duplicate Relation

Unlike the ATDI algorithm, the pure MDS algorithm requires comparison in all facets to isolate update descriptors. Several techniques can be used with this algorithm to improve change detection performance including the introduction of secondary indexes and restricting the size of the duplicate relation to contain only those attributes and tuples that have relevant triggers or snapshots defined against them. The tests run against the MDS algorithm clearly show the different effects of restricting attributes as apposed to restricting tuples. If CA-INGRES provided a full-outer-join operator, the results of these tests could have been improved. As Informix, Oracle, and DB2 all provide interfaces to extend the capabilities of their systems, the performance of the MDS algorithm on these systems may be slightly better[Info97b]. Microsoft Access already provides the SQL3 outer join operator, and SQL3 is gaining momentum, so improvement in this area for other systems is highly likely in the near future. Figure 26 shows the performance characteristics of the MDS algorithms when varying the number of changes performed between iterations as well as varying the size of the duplicate relation both horizontally and vertically.

Unlike ATDI, the differencing query cannot be simplified through the introduction of delete flags and timestamps. Without the introduction of secondary indexes or restricting the size of the replica, the best performance that can be expected from this algorithm is approximately 2N when a sort-merge join with outer join detection is used against presorted relations. The actual results are slightly better than this due to

the benefit of data caching within the server. The prevalent use of the B-tree storage structure in most database products makes this scenario fairly likely.

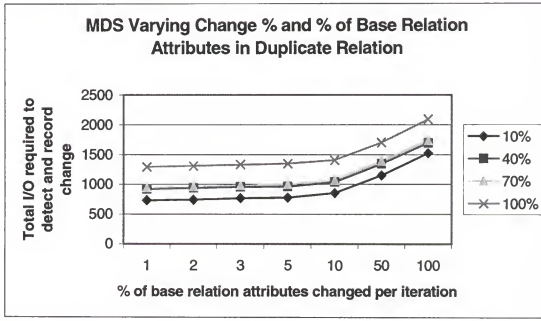


**Figure 26: Performance of MDS Algorithm Restricting Replica Tuples and the Number of Changes Applied between Iterations**

It is often the case, however, that only certain types of changes are of interest. Maintaining non-referenced attributes and tuples in the duplicate relation only serves to increase I/O costs without adding benefit. With knowledge about the type of restrictions desired, the size of the replica can be restricted to dramatically improve the performance of the MDS differencing algorithm.

Figure 26 outlines the affect of changing progressively larger percentages of the base relation when the replica is restricted to a percentage of the base relation's tuples. For these tests, relations with 10,000 rows spanning 1560 pages were used. The duplicate relation is restricted on different attributes. As an example, the replica may be required to maintain information about a single department instead of all departments. If the restriction is highly selective, the entire duplicate relation may be able to fit into memory.

With this simple restriction, the I/O costs for MDS can be reduced to N as 0 IO are required for the replica. In addition, if the base relation has appropriate indexes defined against both the primary key and the attribute(s) that are used to restrict the replica, I/O costs can be further reduced by eliminating the need to scan the base relation to complete the full outer-join emulation. To detect inserts, not all inserts but only those of interest, a secondary index on the restrictive attribute is used. The secondary index is used both in the join to detect changes as well as in the left and right outer joins that are used to detect inserts and deletes. Presuming the existence of a moderately sized cache and a table that is not excessively large, the entire secondary index could remain in memory for the duration of the joined query. Figure 26 clearly shows that a highly restrictive duplicate relation can yield excellent performance even when the entire base relation has been changed since the last iteration of the algorithm.

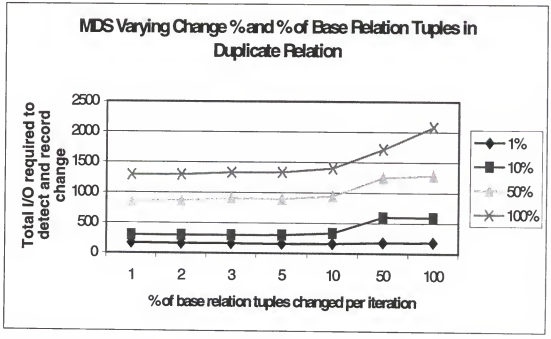


**Figure 27: Performance of MDS Algorithm Restricting Attributes of the Replica and the Number of Changes Applied between Iterations**

When the size of the replica is less restrictive, the MDS algorithm still performs well in the general case where there are a relatively small number of changes between iterations of the algorithm. If the base relation is not structured upon the attribute to which the restrictive condition applies, matching greater than 10% of the tuples will most likely require touching the majority of pages in the base relation [Yao77]. If the base relation can be restructured to group on the restrictive attribute, or if a secondary index can be augmented with applicable attributes, if only a few attributes are of interest, the number of IO required to detect a change is greatly diminished. The improvement is accomplished by essentially comparing two subsets of the data instead of requiring access to the entire data set. If each page of the base relation must be accessed, it is better to use a sort-merge join with outer join detection if the two relations are already sorted. Even if every page of the base table is referenced, less than 2N IO are required to determine changes as the size of the indexes are usually less than .1N and they are often small enough to fit into memory. Thus the total cost for less restrictive replicas is approximately 1.2N in the general case, and 2N in the worst case if both are stored as b-tree structures and the sort-merge join with outer join detection is used.

It is interesting to note in the comparison of Figures 26 and 27 that restricting the number of tuples instead of attributes yields significantly different results when a highly restrictive condition is applied but the results are consistent when the size of the replica approaches that of the base relation. When the tuples in the duplicate relation are restricted to 1% of the base relation, I/O requirements are minimal, regardless of the number of changes in the base relation. Restricting the number of attributes to 1% still

requires accessing every page of the base relation, thus the attribute restriction does not have the magnitude of impact as the tuple restriction. As the restrictions become less, the I/O required by each algorithm slowly merges. This merging of values is due to the use of secondary indexes to assist in identifying changes. The size of the secondary index on the base relation is that same for both types of restrictions. When the number of tuples in the replica is small, the number of pages of the index and the base relation that are accessed is also minimal. For the restriction of attributes, all pages of the base relation must still be accessed unless the secondary is augmented with additional attributes. As the replicas become large, the entire base relation, in both algorithms, must be scanned, thus leading to a similar curve. In our tests, the results are significantly better than the 2N that might be expected due to the benefit of caching within the server.

Performance Summary

The implementation results provide reasonable confirmation of the results outlined in chapter 4. If the underlying data sources provide a queriable interface, our algorithms can be effectively utilized to gather update descriptors without dramatically affecting system performance. This is especially true of the ATDI algorithms which can be used to gather update descriptors at a cost of 0 IO, in many cases, without appreciably lengthening the cost of basic inserts and deletes. The MDS algorithm yields reasonable results if the sizes of the participating relations are not excessive. The performance of the MDS algorithm can be greatly improved by using secondary indexes if the underlying data source supports such a mechanism. The drawback of utilizing a secondary index is

that the cost of basic inserts, deletes, and some updates will be increased due to the necessity of modifying the data in the index structure.

As the performance of the different algorithms is well understood, it may be possible to automatically modify the data source schema and data gathering technique to make the best use of the available techniques. For scenarios where the replica cannot be maintained at the source site, a distributed optimizer with weighting factors for each system could be used to either automatically change or to suggest changes to the data source administrator. In both cases, the performance of these algorithms is a vast improvement over previous techniques and the performance characteristics are sufficient for use with an asynchronous trigger processor or for propagation of data to a data warehouse.

**Table 5: Qualitative Comparison of Update Gathering Techniques with Respect to Selected Relevant Issues**

| | IO Cost | Provides OLTP Support | Widely Available | Space Complexity | Message Cost | Transaction Support | Notification Latency | Frequent Diff Support | Catch Every Update | Selected change Only? | Good For Large Tables | Good for 1%change | Good for 30% change | Support for View Diff |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Replication | 2 | 1 | 5 | 2 | 1 | 2 | 2 | 0 | 2 | 1 | 1 | 1 | 1 | 5 |
| Triggers | 1 | 2 | 4 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 4 | 4 |
| ODBC UPD trap | 1 | 1 | 5 | 1 | 2 | 1 | 1 | 0 | 3 | 2 | 1 | 1 | 1 | 4 |
| Flat File Diff | 5 | 4 | 1 | 4 | 4 | 0 | 5 | 4 | 5 | 5 | 4 | 4 | 4 | 5 |
| TS | 3 | 5 | 2 | 3 | 1 | 0 | 4 | 4 | 4 | 1 | 4 | 3 | 3 | 3 |
| TDI | 2 | 3 | 3 | 3 | 1 | 0 | 3 | 2 | 3 | 1 | 1 | 1 | 3 | 1 |
| ID | 3 | 2 | 2 | 3 | 2 | 0 | 4 | 2 | 4 | 1 | 3 | 3 | 3 | 2 |
| ID/RCS | 3 | 2 | 2 | 3 | 3 | 0 | 4 | 2 | 4 | 2 | 2 | 2 | 4 | 4 |

Given the number of techniques for gathering changes, it would beneficial to create a decision tree to isolate the one appropriate solution for a given situation. The problem with providing a decision tree is that many of the issues determining the "best" algorithm for a given system depends upon weighting factors that an individual user can assign to different components. As a pure decision tree is not possible, we provide a table outlining the most pertinent issues with a ranking between 1 and 5 with 1 being the best, 5 being the worst, and 0 indicating that the given issue does not apply to a particular algorithm. By assigning weighting factors to each category, a DBA can select the best algorithm for a given situation. In general, however, a few basic principals can be adopted.

If a data source supports partitioned replication to a format that is readily available to other systems, then replication is probably the best solution. While triggers capture every change, they also increase the length of every transaction that accesses a given relation. ODBC update trapping is good for row level changes, but not for bulk updates that do not read the entire data set into the application. Also, this technique is not widely available. Flat file differencing is seldom the best technique due to the overhead required. Basic TS is great for insert-only systems, but not for systems that utilize large tables and that allow deletes. TDI is a near optimal approach, but it requires application modification to propagate data to the delete flag field. If you must use differencing on large tables with OLTP support, then ID is the best choice as it does not acquire locks for extended periods of time. Finally, ID/RCS is the only differencing algorithm available if sufficient space is not available on a given machine to support two entire copies of the data set.

# CHAPTER 6
# CONCLUSION

The Update Gathering Architecture proposed here solves at least one key issue that previously was reported to be unsolvable, namely that the excessive IO required by an external triggering mechanism eliminates its viability as a commercial product. [Wid96a][Ston97]. To enable effective triggering mechanisms outside of a DBMS, efficient algorithms are required to match update descriptors to appropriate rule conditions. Efficient matching does not ensure the feasibility of an external trigger system. Update descriptors must be detected and reported in such a way as to eliminate latency, IO, CPU, and locking constraints. In addition, the efficient detection of modifications is of limited use if the update descriptors cannot be assured of delivery to the appropriate destination. Detecting changes in a few well-known sources is of limited use as the number of different data sources used by a single company has grown dramatically in recent years.

We believe that, based upon the algorithms and the architecture presented in this dissertation, an Update Gathering Engine could be developed to solve these issues in a manner that is not only useful in external trigger systems, but is also readily adaptable for use with data warehouses as well. The algorithms outlined dramatically reduce the IO required for detecting changes, thus eliminating the IO issues outlined by Stonebraker [Ston97]. The incremental nature of several of the algorithms eliminates locking contention when working with large relations. The introduction of a Helper process on

the data source machine as well as the introduction of persistent queues ensures delivery of update descriptors. Finally, the use of ODBC compliant SQL enables the algorithms to run, without change, against over 130 different sources of data. The combination of all of these components ensures a readily implementable, highly efficient update gathering engine.

The existence of active and replication facilities in most major DBMS products attests to their usefulness for many business systems. As it is difficult, then, to demean the benefit of their functionality, the unavailability of these capabilities in many other sources of data can most likely be attributed to the complexity of implementation. Even in systems where replication and active capabilities are available, these techniques have not been readily adopted to propagate data into data warehouses due to the complexity of heterogeneous database connectivity from within a single data source.

The work outlined in this paper provides a solution for both of these issues by providing efficient mechanisms to isolate update descriptors and by outlining an architecture to deliver, match, and forward changes to other sources. The TriggerMan project provides the foundation for an external trigger processor that can efficiently match update descriptors against defined triggers. The data gathering architecture enables efficient detection and propagation of update descriptors as well as semantically correct application of trigger actions. The two pieces effectively enable the creation of an external trigger processor that can be used with most structured and some semi-structured sources of data. The extensible design of the architecture allows for the introduction of new data types and update gathering techniques that may be more efficient in a specific context. With the work introduced here, all ODBC compliant data sources can be

augmented with active and replication capabilities without requiring that the underlying system be rewritten.

LIST OF REFERENCES


[ACT96] ACT-NET Consortium, "The Active Database Management System Manifesto: A Rulebase of DBMS Features," *Communications of the ACM*, 25(3), September 1996, Pages 40-49.

[Alon90] Alonso, Rafael, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System," *Transactions on Database Systems (TODS)*, 15(3), September 1990, Pages 359-384.

[BeaT99] "BEA Tuxedo," http://www.beasys.com.

[Bern90a] Bernstein, P., "TP Monitors," *Communications of the ACM*, 33(11), November 1990, Pages 75-86.

[Bern90b] Bernstein, P., M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues," *Communications of the ACM*, 19(2), November 1990, Pages 112-122.

[Bern91] Bernstein, P., W. Emberton, and V. Treban, "DECdta – Digital's Distributed Transaction Processing Architecture," *Digital Technical Journal*, 3(1), Winter 1991, Pages 1-16.

[Bran93] D.A. Brant and D.P. Miranker, "Index Support for Rule Activation," *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993, Pages 42-48.

[Brow97] Brown, L., *Oracle Database Administration*, Prentice Hall Publishers, Upper Saddle River, NJ, 1997.

[Chak89] Chakravarthy, S., "Rule Management and Evaluation: An Active DBMS Perspective," SIGMOD Record, 18(3), March, 1989, Pages 20-28.

[Chak91] Chakravarthy, S., "Active Database Management Systems: Requirements, State-of-the-art, and an Evaluation," *E-R Approach: the Core of Conceptual Modeling*, Elsevier Science, Amsterdam, 1991.

[Chaw96] Chawathe, S., A. Rajaraman , H. Garcia-Molina, and J. Widom, "Change Detection in Hierarchically Structured Information," *Proceedings ACM International Conference on the Management of Data*, 25(2), June, 1996,

Pages 493-50.

[Chaw97a] Chawathe, S., S. Abiteboul, J. Widom, "Representing and Querying Changes in Semistructured Data," *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, February 1997, Pages 4-13.

[Chaw97b] Chawathe, S., H. Garcia-Molina, "Meaningful Change Detection in Structured Data," *Proceedings of the ACM International Conference on the Management of Data*, 26(2), June 1997, Pages 26-37.

[Come91] Comer, Douglas E., *Internetworking with TCP/IP,* 2nd Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Comp94] Computer Associates, *CA-OpenIngres/Replicator*, CA International, Inc., Alameda, CA, 1994.

[Date93] Date, C.J., and Hugh Darwen, *A Guide to the SQL Standard,* 3rd Edition, Addison Wesley, Reading, MA, 1993.

[Dewa92] H.M. Dewan, D. Ohsie, S.J. Stolfo, O. Wolfson, and S. Da Silva, "Incremental Database Rule Processing in PARADISER," *Journal of Intelligent Information Systems,* 1(2), 1992, Pages 177-209.

[Dunl90] Dunlop, N., *DBASE for Professionals,* Van Nostrand Reinhold, New York, 1990.

[Enci98] Encina, Developing Distributed Transaction Applications with Encina, http://www.redbooks.ibm.com/index.html, December 7, 1998.

[Etzi93] Etzion, O., "PARDES: A Data-driven Oriented Active Database Model," *SIGMOD Record*, 22(1), Pages 7-14, March 1993.

[Felb96] Felber, P., B. Garbinato, and R. Guerraoui, "The Design of a CORBA Group Communication Service," Symposium on Reliable Distributed Systems, Niagra-on-the-lake, Canada, 1996, Pages 150-159.

[Geig95] Geiger, K., *Inside ODBC*, Microsoft Press, Redmond, WA, 1995.

[Gray93] Gray, Jim, *Transaction Processing, Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[Guer95a] Guerraoui, R., A. Schiper, "A Generic Multicast Primitive to Support Transactions on Replicated Objects in Distributed Systems," *Proceedings of the IEEE Computing Society, Workshop on Future Trends in Distributed Computing Systems*, Cheju Island, Korea, August 1995, Pages 334-342.

[Guer95b] Guerraoui, R., M. Larrea, A. Schiper, "Non Blocking Atomic Commitment with an Unreliable Failure Detector," *IEEE Symposium on Reliable Distributed Systems*, Bad Neuenahr, Germany, September 1995, Pages 41-50.

[Guer96a] Guerraoui, R., M. Larrea, A. Schiper, "Reducing the Cost for Non-Blocking in Atomic Commitment," *International Conference on Distributed Computing Systems*, Hong Kong, May 1996, Pages 692-697.

[Guer96b] Guerraoui, R., A. Schiper, "The Decentralized Non-Blocking Atomic Commitment Protocol," *IEEE Symposium on Parallel and Distributed Processing*, San Antonio, TX, October 1996, Pages 2-9.

[Hans92] Hanson, E., "Rule Condition Testing and Action Execution in Ariel," *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA, June 1992, Pages 49-58.

[Hans94] Hanson, E., and T. Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," *Tech Report TR94-017*, CISE Department, University of Florida, http://www.cise.ufl.edu, October 1994.

[Hans95] Hanson, E. and J. Widom, "Rule processing in active database systems," *Advances In Databases and Artificial Intelligence*, 1, 1995, Pages 33-77.

[Hans96a] Hanson, E., "The Design and Implementation of the Ariel Active Database Rule System," *IEEE Transactions on. Knowledge and Data Engineering*, 8(1), February 1996, Pages 157-172.

[Hans96b] Hanson, E., S. Bodagala, M. Hasan, G. Kulkarni, and J. Rangarajan, "Optimized Rule Condition Testing in Ariel Using Gator Networks," *TR 95-027*, CISE Department, University of Florida, http://www.cise.ufl.edu, October 1995.

[Hans97c] Hanson, E., and Samir Khosla, "An Introduction to the TriggerMan Asynchronous Trigger Processor," *Proceedings of the 3rd International Workshop on Rules in Database Systems*, Skovde, Sweden, June 1997.

[Info97a] "Informix Universal Server," http://www.informix.com, 1998.

[Info97b] Informix, "Developing DataBlade Modules for INFORMIX-Universal Server," http://www.informix.com, 1998.

[Ingr91] Ingres, "Ingres SQL Reference Manual," Ingres Corporation, Alameda, CA, 1991.

[Jaeg95] U. Jaeger, and J. C. Freytag, "Annotated Bibliography on Active Databases," *SIGMOD Record*, 24, 1995, Pages 58-69.

[Kawa97] Kawaguchi, Akira, D. Lieuwen, I. Mumick, D. Quass, K. Ross, "Concurrency Control Theory for Deferred Materialized Views," *Proceedings of the International Conference on Database Theory*, June, 1997.

[Kudr93] T. Kudrass, H. Branding, A. Buchmann, and J. Zimmerman, "Rules in an Open System: The REACH Rule System," *In Proceedings of the 1st International Workshop on Rules in Database Systems*, Edinburg, Scotland, August 1993, Pages 111-126.

[Labi96] Labio, W., and H. Garcia-Molina, "Efficient Snapshot Differential Algorithms for Data Warehousing," *Proceedings of VLDB*, Mumbai, India, September 1996, Pages 63-74.

[Labi97] Labio, W., D. Quass, and B. Adelberg, "Physical Database Design for Data Warehouses," *Proceedings of the 13th International Conference on Data Engineering*, Birmingham, UK, April 1997, Pages 277-288.

[Lamb91] Lamb, C., G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System," *Communications of the ACM*, 34(10), October 1991, Pages 50-63.

[Liao97] Liao, H., "Global Events in Sentinel: Design and Implementation of a Global Event Detector," Masters Thesis, University of Florida, 1997.

[Lind86] Lindsay, B., L. Haas, C. Mohan, H. Pirahesh, P. Wilms, "A Snapshot Differential Refresh Algorithm," *Communications of the ACM*, 15(2), 1986, Pages 53-60.

[McCa89] McCarthy, Dennis R. and U. Dayal, "The Architecture of an Active Data Base Management System," *In Proceedings of the. ACM SIGMOD Conference on Management of Data.*, Portland, OR, June, 1989, Pages 215-224.

[Micr97] Microsoft, *Microsoft SQL Server*, Microsoft, Redmond, WA, 1996.

[Mois94] Moissis, A. "Sybase Replication Server," Sybase Technical Paper Series, Emeryville, CA, 1994.

[ODBC94] ODBC 2.0 Programmer's Reference and SDK Guide, Microsoft Press, Redmond, WA, 1994.

[Orac94] Oracle, "Oracle 7 Symmetric Replication," Oracle White Paper, June 1994.

[Oren92] Orenstein, J., S. Haradhvala, B. Margulies, and D. Sakahara, "Query Processing in the ObjectStore Database System," *Proceedings of the ACM SIGMOD*, San Diego, CA, June 1992, Pages 403-412.

[Penn97] Penner, R., "ODBC 3.0: What's New?," Unix Review, April 1997.

[Quas96] Quass, D, A. Gupta, I. Mumick, J. Widom, "Making Views Self-Maintainable for Data Warehousing," *Proceedings of the 14th Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1996, Pages 158-169.

[Rapi96] "RapidBase Technical Summery," http://www.vtt.fi/tte/projects/rapid, February, 1999.

[Roth97] Roth, M.T., P. Schwarz, "Don't Scrap it, Wrap it: A Wrapper Architecture for Legacy Data Sources," *VLDB*, Athens, Greece, August 1997, Pages 266-275.

[Sist95] Sistla, A., and O. Wolfson, "Temporal Triggers in Active Databases," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(3), June 1995, Pages 471-486.

[Skee81] Skeen, D. "Nonblocking Commit Protocols," *Proceedings of the ACM SIGMOD*, Ann Arbor, MI, April 1981, Pages 133-142.

[Ston87] M. Stonebraker, Eric Hanson, and P. K. Chan, "The Design of the POSTGRES Rule System," *In Proceedings of the 1987 IEEE Data Engineering Conference*, Los Angeles, CA, Pages 365-374.

[Ston90] M. Stonebraker, L. Rowe, and M. Hirohama, "The Implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering*, 2(7), March 1990, Pages 125-142.

[Ston97] M. Stonebraker, D. Moore, *Object Relational DBMSs; The Next Great Wave*, Morgan Kaufman, San Mateo, CA, April 1996.

[Syba96] *Sybase Replication Server Technical Overview*, Sybase Inc., Emeryville, CA, 1996.

[TPCD98] Transaction Processing Performance Council (TPC), "TPC Benchmark D," TPC Standards Specification, 1998, http://www.tpc.ord/dspec.html.

[Ullm89] Ullman, J.D., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, MD, 1989.

[Usof96] Usoft, "Business Rules Automation," Usoft White Paper, 1996, http://www.usoft.com/whitepapers.

[Wido92] J. Widom, "The Starburst Rule System: Language Design, Implementation, and Applications," *IEEE Data Engineering*, 15, 1992, Pages 1-4.

[Wido96a] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules For Advanced Database Processing,* Morgan Kaufmann, San Mateo, CA, 1996.

[Wido96b] J. Widom, "The Starburst Active Database Rule System," *IEEE Transactions on Knowledge and Data Engineering*, 8, 1996, Pages 583-595.

[Wols96] Wolski, A., J. Karvonen, A. Puolakka, "The RAPID Case Study: Requirements for and the Design of a Fast-Response Database System," *Proceedings of the First Workshop on Real-Time Databases*, March 1996.

[Yao77] Yao, S., "Approximating Block Accesses in Database Organizations," *Communications of the ACM*, 20(4), April 1977, Page 260.

[Zhug96a] Zhuge, Y., H. Garcia-Molina, and J. Wiener, "The Strobe Algorithms for Multi-Source Warehouse Consistency," *Proceedings of the Conference on Parallel Distributed Information Systems*," Miami Beach, FL, December 1996.

[Zhug96b] Zhuge, Y., H. Garcia-Molina, and J. Wiener, "Consistency Algorithms for Multi-Source Warehouse View Maintenance" *Journal of Distributed and Parallel Databases*, 6(1), July 1997, Pages 7-40.

## BIOGRAPHICAL SKETCH

Tony Christopher Carnes was born in Prince Edward County, Virginia in 1964. He attended Hampden-Sydney College from 1982 until December of 1985 where he earned his Bachelor of Science degree in math and computer Science. While at HSC, he participated in varsity football, track, and cross-country. He was captain of the cross-country and track teams in 1985. Immediately after college, Chris worked for the Foreign Mission Board of the Southern Baptist Convention for four years. During that time he attended classes at the University of Richmond in their Master of Business Administration program. While at UR, he ran track and coached the UR women's volleyball team. In 1990, he left the Foreign Mission Board and started Carnes Consulting. For the next three years, he traveled extensively throughout the United States both lecturing and consulting. In 1993, after getting married, he returned to school at the University of Florida. In May of 1996, he earned his Master of Science degree in computer and information science and engineering.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Eric N. Hanson, Chairman
Associate Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Upendranatha Chakravarthy
Associate Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Richard Newman
Assistant Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Beverly Sanders
Associate Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Suleyman Tufekci
Associate Professor of Industrial and Systems Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May, 1999

Winfred M. Phillips
Dean, College of Engineering

M. J. Ohanian
Dean, Graduate School